Introduction to Software Development (ISD)
Week 3

Autumn term 2012

londonknowledgelab

Aims of Week 3

- To learn about `while`, `for`, and `do` loops
- To understand and use nested loops
- To implement programs that read and process data sets
- To write and use methods

The `while` loop

- Compound interest algorithm: Suppose you put £10,000 into a bank account that earns 5 percent interest per year. How many years does it take for the account balance to be double the original?
- Pseudocode that defines an algorithm to solve this problem:

```
year = 0;
balance = 10000;
while (balance < 20000) {
  year = year + 1;
  interest = balance * 0.05;
  balance = balance + interest;
}
output year value
```

In Java:

```
1  /**
2     This program computes the time required to double an investment.
3  */
4  public class DoubleInvestment
5  {
6     public static void main(String[] args)
7     {
8        final double RATE = 5;
9        final double INITIAL_BALANCE = 10000;
10       final double TARGET = 2 * INITIAL_BALANCE;
11
12       double balance = INITIAL_BALANCE;
13       int year = 0;
14
15       // Count the years required for the investment to double
16
17       while (balance < TARGET)
18       {
19          year++;
20          double interest = balance * RATE / 100;
21          balance = balance + interest;
22       }
23
24       System.out.println("The investment doubled after "
25          + year + " years.");
26    }
27 }
```

Note about the ‘scope’ of variables

- Declaring a variable *inside* a loop body, e.g. as with the variable `interest` in the previous program, means that a new variable is created each time and is removed at the end of each iteration round the loop

- Declaring a variable *before* a loop body, e.g. as with the variables `balance` and `year`, means that these same variables are used for each iteration round the loop.
  - They retain their current values on each iteration round the loop
  - They retain their current values after the end of the loop and can be used after the loop

What is wrong with this code snippet:

```
int year = 0;
double balance = 10000;
while (year < 20) {
  double interest = balance * 0.05;
  balance = balance + interest;
}
```

What is wrong with this code snippet:

```
int year = 0;
double balance = 1000;
while (year < 20) {
  double interest = balance * 0.05;
  balance = balance + interest;
}
```

Answer: the loop never ends. The program will “hang” and will need to be stopped by the user (check out how to stop programs in our IDE)

What is wrong with this code snippet:

```
int year = 20;
double balance = 10000;
while (year > 0) {
  year = year + 1;
  double interest = balance * 0.05;
  balance = balance + interest;
}
```

What is wrong with this code snippet:

```
int year = 20;
double balance = 10000;
while (year > 0) {
  year = year + 1;
  double interest = balance * 0.05;
  balance = balance + interest;
}
```

Answer: the loop never ends. The programmer probably meant to write `year = year - 1;`

What is wrong with this code snippet:

```
int year = 20;
double balance = 10000;
while (year > 0) ; {
  year = year - 1;
  double interest = balance * 0.05;
  balance = balance + interest;
}
```

What is wrong with this code snippet:

```
int year = 20;
double balance = 1000;
while (year > 0) ; {
  year = year - 1;
  double interest = balance * 0.05;
  balance = balance + interest;
}
```

Answer: the loop never ends – due to the ; which is understood by the compiler as being the body of the loop

More `while` loop examples

- Averaging a set of non-negative numbers, ended by the user entering the ‘sentinel’ value -1:

```
 8    public static void main(String[] args)
 9    {
10       double sum = 0;
11       int count = 0;
12       double salary = 0;
13       System.out.print("Enter salaries, -1 to finish: ");
14       Scanner in = new Scanner(System.in);
15
16       // Process data until the sentinel is entered
17
18       while (salary != -1)
19       {
20          salary = in.nextDouble();
21          if (salary != -1)
22          {
23             sum = sum + salary;
24             count++;
25          }
26       }
27
```

```
28          // Compute and print the average
29
30          if (count > 0)
31          {
32             double average = sum / count;
33             System.out.println("Average salary: " + average);
34          }
35          else
36          {
37             System.out.println("No data");
38          }
39       }
40    }
```

Program Run

```
Enter salaries, -1 to finish: 10 10 40 -1
Average salary: 20
```

Averaging a set of numbers that may be positive or negative numbers:

- We cannot use -1 (or any other number) as the sentinel!
- We can use a non-numeric sentinel value, such as “Q”
  - But `in.nextDouble` will fail when “Q” is input by the user
  - So use `in.hasNextDouble` to first test whether the next input is a number
  - If `in.hasNextDouble` returns `true` then we can safely use `in.nextDouble` to read the next input into a double:

```
System.out.print("Enter values, Q to quit: ");
while (in.hasNextDouble()) {
   double value = in.nextDouble();
   . . . // Process value
}
```

Summing or Averaging a set of numbers:

```
double total = 0;
while (in.hasNextDouble()) {
  double input = in.nextDouble();
  total = total + input;
}
```

```
double total = 0;
int count = 0;
while (in.hasNextDouble()) {
  double input = in.nextDouble();
  total = total + input;
  count++;
}
double average = 0;
if (count > 0) {
  average = total / count;
}
```

finding the Maximum or Minimum of a set of numbers:

```
double largest = in.nextDouble();
while (in.hasNextDouble()) {
  double input = in.nextDouble();
  if (input > largest) {
    largest = input;
  }
}
```

```
double smallest = in.nextDouble();
while (in.hasNextDouble()) {
  double input = in.nextDouble();
  if (input < smallest) {
    smallest = input;
  }
}
```

- Get first input value
  - This is the largest (or smallest) that you have seen so far
- Loop `while` you have a valid number (non-sentinel)
  - Get another input value
  - Compare new input to largest (or smallest)
  - Update largest (or smallest) if necessary

Comparing Consecutive Values for duplicates:

```
double input = in.nextDouble();
while (in.hasNextDouble()) {
  double previous = input;
  input = nextDouble();
  if (input == previous) {
    System.out.println("Duplicate input");
  }
}
```

- Get first input value
- Use `while` to determine if there are more to check
  - Copy input to previous variable
  - Get next value into input variable
  - Compare input to previous, and output if same

Finding the First Match in a string

To find the position of the first lower-case letter in a string, str:

```
boolean found = false;
char ch;
int position = 0;
while (!found && position < str.length()) {
  ch = str.charAt(position);
  if (Character.isLowerCase(ch)) {
    found = true;
  }
  else {
    position++;
  }
}
```

The for loop

- Sometimes we want to execute a sequence of statements a *known* number of times e.g. to compute the balance after 20 years, with an initial amount of £10000 and 5% annual interest:

```
int year = 0;
double balance = 10000;
while (year < 20) {
  year++;
   double interest = balance * 0.05;
  balance = balance + interest;
}
System.out.printf(“%s%10.2f”,
  ”Balance after 20 years is ”, balance);
```

- This can be expressed more succinctly using a `for` loop:

The for loop

```
double balance = 10000;
for (int year = 0; year < 20; year++) {
  double interest = balance * 0.05;
  balance = balance + interest;
}
System.out.printf("%s%10.2f",
  "Balance after 20 years is ", balance);
```

- `int year = 0` is the initialisation and happens once, before the loop starts
- `year < 20` is the condition which is checked before each iteration of the loop
- `year++` is executed after each iteration of the loop

‘Scope’ of variables

- In the below, the variable `year` cannot be used after the loop:

```
double balance = 10000;
for (int year = 0; year < 20; year++) {
  double interest = balance * 0.05;
  balance = balance + interest;
}
```

- To be able to do that, we need to declare `year` before the loop:

```
int year;
for (year = 0; year < 20; year++) {
        ...
}
... year ...
```

Number of iterations

```
double balance = 10000;
for (int year = 0; year < 20; year++) {
  double interest = balance * 0.05;
  balance = balance + interest;
}
```

- has the same effect as:

```
double balance = 10000;
for (int year = 1; year <= 20; year++) {
  double interest = balance * 0.05;
  balance = balance + interest;
}
```

for loop examples

Table 2 for Loop Examples

Loop	Values of i	Comment
`for (i = 0; i <= 5; i++)`	0 1 2 3 4 5	Note that the loop is executed 6 times. (See Programming Tip 4.4 on page 153.)
`for (i = 5; i >= 0; i--)`	5 4 3 2 1 0	Use `i--` for decreasing values.
`for (i = 0; i < 9; i = i + 2)`	0 2 4 6 8	Use `i = i + 2` for a step size of 2.
`for (i = 0; i != 9; i = i + 2)`	0 2 4 6 8 10 12 14 ... (infinite loop)	You can use < or <= instead of `!=` to avoid this problem.
`for (i = 1; i <= 20; i = i * 2)`	1 2 4 8 16	You can specify any rule for modifying `i`, such as doubling it in every step.
`for (i = 0; i < str.length(); i++)`	0 1 2 . . . until the last valid index of the string `str`	In the loop body, use the expression `str.charAt(i)` to get the `i`th character.

Traversing all characters of a string:

```
String str = in.next();
for (int i = 0; i < str.length(); i++) {
  char ch = str.charAt(i);
  . . . // Process ch
}
```

Counting matches in a string:

To find the number of upper-case letters in a string, str:

```
int upperCaseLetters = 0;
for (int i = 0; i < str.length(); i++) {
  char ch = str.charAt(i);
  if (Character.isUpperCase(ch)) {
    upperCaseLetters++;
  }
}
```

The do loop

- Sometimes we want to execute the loop body at least once, and test the condition after the first iteration
- The do loop is useful for this e.g. to validate that a user has entered an integer less than 100, we can write:

```
int value;
do {
  System.out.println("Enter an integer < 100: ");
  value = in.nextInt();
}
while (value >= 100);
```

Nested loops

- How would you print a table with rows and columns?
  - Print the table header using one or more `for` loops
  - Print the table body:
    - loop per row
    - and for every row loop per column

```
1  /**
2     This program prints a table of powers of x.
3  */
4  public class PowerTable
5  {
6     public static void main(String[] args)
7     {
8        final int NMAX = 4;
9        final double XMAX = 10;
10
11       // Print table header
12
13       for (int n = 1; n <= NMAX; n++)
14       {
15          System.out.printf("%10d", n);
16       }
17       System.out.println();
18       for (int n = 1; n <= NMAX; n++)
19       {
20          System.out.printf("%10s", "x ");
21       }
22       System.out.println();
```

```
         1         2         3         4
         x         x         x         x

         1         1         1         1
         2         4         8        16
         3         9        27        81
         4        16        64       256
         5        25       125       625
         6        36       216      1296
         7        49       343      2401
         8        64       512      4096
         9        81       729      6561
        10       100      1000     10000
```

```
  1          2          3          4
  x          x          x          x

  1          1          1          1
  2          4          8         16
  3          9         27         81
  4         16         64        256
  5         25        125        625
  6         36        216       1296
  7         49        343       2401
  8         64        512       4096
  9         81        729       6561
 10        100       1000      10000
```

```
24          // Print table body
25
26          for (double x = 1; x <= XMAX; x++)
27          {
28             // Print table row
29
30             for (int n = 1; n <= NMAX; n++)
31             {
32                System.out.printf("%10.0f", Math.pow(x, n));
33             }
34             System.out.println();
35          }
36       }
37    }
```

More Nested Loop Examples

```
for (i = 1; i <= 3; i++)
{
   for (j = 1; j <= 4; j++)  { print "*" }
   print new line
}
```

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

Prints 3 rows of 4 asterisks each.

```
for (i = 1; i <= 4; i++)
{
   for (j = 1; j <= 3; j++) { print "*" }
   print new line
}
```

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

Prints 4 rows of 3 asterisks each.

```
for (i = 1; i <= 4; i++)
{
   for (j = 1; j <= i; j++) { print "*" }
   print new line
}
```

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

Prints 4 rows of lengths 1, 2, 3, and 4.

More Nested Loop Examples

```
for (i = 1; i <= 3; i++)
{
   for (j = 1; j <= 5; j++)
   {
      if (j % 2 == 0) { print "*" }
      else { print "-" }
   }
   print new line
}
```

```
-*-*-
-*-*-
-*-*-
```

Prints asterisks in even columns, dashes in odd columns.

```
for (i = 1; i <= 3; i++)
{
   for (j = 1; j <= 5; j++)
   {
      if ((i + j) % 2 == 0) { print "*" }
      else { print " " }
   }
   print new line
}
```

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

Prints a checkerboard pattern.

Summary of loops

- There are three types of loops in Java (and in other similar programming languages):
  - **`while`** Loops
  - **`for`** Loops
  - **`do`** Loops
- Each loop requires the following steps:
  - Initialisation (get ready to start looping)
  - Condition (test if we should execute the loop body)
  - Update (change something each time round the loop)

Methods

- A ***method*** is a sequence of instructions with a name
- We ***call*** a method in order to execute its instructions
- We should not have to know how a method is implemented in order to use it:
  - we should be able use it as a `black box' from its specification only
- One method can call another method
- Some Java library methods that we have already used are Math.pow(), String.length(), Scanner.nextInt() etc.

Calling Methods

- For example:

```
public static void main(String[] args) {
  . . .
  double z = Math.pow(2, 3);
  . . .
}
```

- Here, the `main` method calls the `Math.pow` method with the inputs 2 and 3 – these are termed the *parameter values* or *arguments* that are input to the method when it is called

- The result returned by `Math.pow` – termed its *return value* – is assigned to the variable `z`

Implementing Methods

- Suppose we want to implement a method to calculate the volume of a cube. We need to think about:
  - What input does it need to do its job?
  - What answer does it return?

- When writing the method:
  - Pick a meaningful name for the method (`cubeVolume`).
  - Give a type and a name for each parameter (`double sideLength`)
  - Specify the type of the return value (`double`)
  - Add the appropriate “modifiers”, such as `public static` (for now, all the methods that we write will be of this kind, which are known as *static methods*)

```
public static double cubeVolume(double sideLength)
```

Inside the ‘box’

- Then we need to design and write the body of the method
  - The body is surrounded by curly braces `{ }`
  - The body contains the variable declarations and statements that are executed when the method is called
  - It will also return the answer computed by the method
  - Below, `double sideLength` is termed a *parameter variable* or *formal parameter* and it can be used in the body of the method:

```
public static double cubeVolume(double sideLength)
{
  double volume = sideLength * sideLength * sideLength;
  return volume;
}
```

Program using the cubeVolume Method:

```
1  /**
2     This program computes the volumes of two cubes.
3  */
4  public class Cubes
5  {
6     public static void main(String[] args)
7     {
8        double result1 = cubeVolume(2);
9        double result2 = cubeVolume(10);
10       System.out.println("A cube with side length 2 has volume " + result1);
11       System.out.println("A cube with side length 10 has volume " + result2);
12    }
13
14    /**
15       Computes the volume of a cube.
16       @param sideLength the side length of the cube
17       @return the volume
18    */
19    public static double cubeVolume(double sideLength)
20    {
21       double volume = sideLength * sideLength * sideLength;
22       return volume;
23    }
24 }
```

Commenting Methods

- When you write a method, you should precede it with a comment that describes its behaviour. This is so that you, and others, can understand its use.

- Start the comment with `/**`
  - Briefly describe the purpose of the method
  - Describe each parameter in a line starting with `@param`
  - Describe the return value in a line starting with `@return`
- End the comment with `*/`

- Note that these comments describe *what* the method does, not *how* it does it i.e. they give its *specification.* The specification of a method should be sufficient to allow other programmers to use it as a ‘black box’, without knowing its implementation.

Modifying parameter variables

- Although parameter variables can be modified inside the body of a method, this can be confusing and it is better to introduce a separate variable. For example, consider this:

```
public static int total(int pounds, int pence)
{
    pence = pounds * 100 + pence;
    return pence;
}
```

- If a variable is passed as the actual parameter to a method and the parameter variable is modified inside the method, the actual parameter itself is *not* changed e.g. calling the above method with

```
int myPounds = 5, myPence = 75;
int totalPence = total(myPounds,myPence);
```

the value of `myPence` is still 75 after the method returns

Return statements

- A return statement in a method body does two things:
  1) The method terminates immediately
  2) The return value is returned to the calling method

- Every branch of a method must have a return statement e.g. the compiler will complain about this:

```
public static double cubeVolume(double sideLength)
{
  if (sideLength >= 0) {
    return sideLength * sideLength * sideLength;
  }
}
```

Return statements

- This is now ok:

```
public static double cubeVolume(double sideLength)
{
  if (sideLength >= 0) {
    return sideLength * sideLength * sideLength;
  }
  else {
    return 0;
  }
}
```

Methods without return values

- Methods do not have to return a value – a return type void can be specified
- No return statement is required in this case;
- But if there is a return statement, then the method stops executing immediately
- E.g. to print a triangle pattern like this:

```
[]
[][]
[][][]
[][][][]
```

- we can write the following method:

```
public static void  printTriangle(int sideLength)
{
  if (sideLength < 1) {
   return;
  }
  for (i = 1; i <= sideLength; i++) {
   for (j = 1; j <= i; j++) {
          System.out.print("[]");
   }
   System.out.println();
  }
}
```

- and we can call it in a statement like this (it does not return a value so it can't be used in an expression or an assignment):

```
printTriangle(4);
```

Scope of variables

- Variables declared inside one method are not visible to other methods
  - In the below, the variable `sideLength` is local to `main`
  - So this will cause a compile-time error:

```
public static void main(String[] args)
{
  double sideLength = 10;
  double result = cubeVolume();
  System.out.println(result);
}

public static double cubeVolume()
{
  return sideLength * sideLength * sideLength; // ERROR
}
```

Scope of variables

- In the below, `result` is local to `square` and `result` is local to `main`
- They are two different variables

```
public static int square(int n)
{
  int result = n * n;
  return result;
}

public static void main(String[] args)
{
  int result = square(3) + square(4);
  System.out.println(result);
}
```

result

result

Recursive Methods

- A method can call itself – these are known as ***recursive methods***
- A recursive algorithm solves a problem by using the solution of the same problem with simpler inputs
- For the algorithm terminate, there must be special cases for ending the computation with the simplest inputs
- For example, here is a recursive method to print a triangle of a given length (the output is the same as with the iterative version of this method that we saw earlier):

```
public static void printTriangle(int sideLength)
{
  if (sideLength < 1) {
     return;
  }
  printTriangle(sideLength - 1);
  for (int i = 0; i < sideLength; i++) {
    System.out.print("[]");
  }
  System.out.println();
}
```

If we start by calling printTriangle(4)
this then calls printTriangle(3)
which then calls printTriangle(2)
which then calls printTriangle(1)
which then calls printTriangle(0)
which returns doing nothing
and then prints []
and then prints [][]
and then prints [][][]
and then prints [][][][]

Aims of Week 3

- To learn about `while`, `for`, and `do` loops
- To understand and use nested loops
- To implement programs that read and process data sets
- To write and use methods

Week 3 Homework

- Complete Lab Sheet 3 – not assessed. Solutions will be posted next week.
- Read Chapters 4 and 5 of *Java for Everyone* and do the self-check questions. Section 4.9 of Chapter 4 is optional reading.
- Make sure you read How To 4.1 (Writing a Loop) and How To 5.1 (Implementing a Method). Also Programming Tip 5.5 (on writing “stub” methods for methods that you haven’t implemented yet)
- If you have time, do some the review and programming exercises from Chapters 4 and 5

Hand-tracing

- Hand-tracing helps you understand whether a program works correctly
- Create a table with one column for each variable
- Use pencil and paper to track their values
- You can hand-trace pseudocode or actual program code
- Use example input values that:
  - You know what the correct outcome should be
  - Will test each branch of your pseudocode
  - Will test boundary values