IMPLEMENTING METHODS

(Chapter 4)

Implementing methods

Introduction

- A method is a part of a class, and contains a particular set of instructions
- So far all the classes you have written have contained just one method, the main method
- Normally, a method will perform a single well-defined task

Examples

- A method could perform to calculate the area of a circle
- A method to display a particular message on the screen
- A method to convert a temperature from Fahrenheit to Celsius

Calling a method

- When we get a method to perform its task we say that we are **calling** the method

- When we call a method, what we are actually doing is telling the program to
  - jump to a new place (where the method instructions are stored)
  - carry out the set of instructions that it finds there
  - when it has finished, return and carry on where it left off

Declaring and defining methods

- Program 4.1 prompts the user to enter his or her first name, family name and town
- Each time the prompt is displayed, it is followed by a message
- We have had to type out the two lines that display the confidentiality message three times
- Instead we could have written a **method**

Program 4.1

```
import java.util.*;

public class DataEntry{
   public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        String firstName, familyName, town;

        System.out.println("First name");
        System.out.println("Note that all information is confidential");
        System.out.println("No personal details will be shared");
        firstName = sc.next();

        System.out.println("Family name");
        System.out.println("Note that all information is confidential");
        System.out.println("No personal details will be shared");
        familyName = sc.next();

        System.out.println("Town");
        System.out.println("Note that all information is confidential");
        System.out.println("No personal details will be shared");
        town = sc.next();
   }
}
```

Declaring and defining methods

```
private static void displayMessage(){
     System.out.println("Note that all information is confidential");
     System.out.println("No personal details will be shared");
}
```

- The body of this method (between the two curly brackets) contains the instructions
- The first line, which declares the method, is called the method **header**

Explanation of words in method header

private

- Placing **private** in front of the method name means that the method cannot be accessed by any other class
- If you wanted the methods of your class to be used by other classes, you would declare your method as **public**
- The method above is here purely to "help" the main method of this class, and so we declare it as **private**
- A **private** method such as this, which is not accessible to other classes, is often referred to as a **helper** method

Explanation of words in method header

static

- The meaning of this will not be explained fully until chapter 7

- For now, you just need to know is that this method has to be **static**, because it is going to be called from another method (that is, the main method) that is also **static**

Explanation of words in method header

void

- It is possible for a method to send back some information once it terminates
- This particular method does not need to do so
- The word **void** indicates that the method does not send back any information

Explanation of words in method header

displayMessage( )

- This is the name that we have chosen to give our method
- It is followed by a pair of empty brackets
- If we want to send information into a method we list, in these brackets, the types of data that we are going to send in
- Here, however, we do not have to send in any data, and the brackets are left empty

```
  private static void displayMessage(){
       System.out.println("Note that all information is confidential");
       System.out.println("No personal details will be shared");
  }
```

Calling a method

- To call a method in Java (i.e. to get it to do its job), we simply use its name, along with the following brackets, which in the above case are empty

```
displayMessage();
```

- We re-write program 4.1, replacing the appropriate lines of code with the simple message call (program 4.2)
- The method is defined separately after the main method
- It could have come before it, since the order in which methods are presented doesn't matter to the compiler
- When the program is run, it always starts with main

Program 4.2

```
import java.util.*;

public class DataEntry{
   public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        String firstName, familyName, town;

        System.out.println("First name");
        displayMessage();
        firstName = sc.next();

        System.out.println("Family name");
        displayMessage();
        familyName = sc.next();

        System.out.println("Town");
        displayMessage();
        town = sc.next();
   }
   private static void displayMessage(){
        System.out.println("Note that all information is confidential");
        System.out.println("No personal details will be shared");
   }
}
```

A reminder of program 1.4

```
import java.util.*;

// a program that calculates and displays the cost
// of a product after tax has been added

public class FindCost3{

  public static void main(String[] args){
      Scanner sc = new Scanner(System.in);
      System.out.print("Initial price: ");
      double price = sc.nextDouble();
      System.out.print("Tax rate: ");
      double tax = sc.nextDouble();

      price = price * (1 + tax / 100);

      System.out.println(“After tax = " + price);
  }
}
```

Method input and output

```
   price = price * (1 + tax/100);
```

- We will create a method that performs this calculation
- We could call this method at various points within the program
- Each time we do so, we can do the calculation for different values of the price and the tax
- We will need a way to send in these values to the method
- We also need to arrange for the method to tell us the result of adding the new tax

Explanation of method header

```
    private static double addTax(double priceIn, double taxIn)
    {
      return priceIn * (1 + taxIn/100);
    }
```

- We are declaring a method of *type* **double**
- The *type* of a method refers to its *return* type
- It is possibly to declare methods of any type - **int**, **boolean**, **char** and so on
- The return type could even be a class such as String
- Within the brackets we are declaring two variables, both of type **double**

Explanation of method header

```
private static double addTax(double priceIn, double taxIn)
{
  return priceIn * (1 + taxIn/100);
}
```

- Variables declared in this way are known as the **formal parameters** of the method
- They are going to hold, respectively, the values of the price and the tax that are going to be sent in from the calling method
- These variables could be given any name we choose, but we have called them priceIn and taxIn respectively

Explanation of method body

```
    private static double addTax(double priceIn, double taxIn)
    {
      return priceIn * (1 + taxIn/100);
    }
```

- **return** ends the method
  - As soon as the program encounters this word, the method terminates, and control of the program jumps back to the calling method
- **return** sends back a value
  - In this case, it sends back the result of the calculation
  - If the method is of type **void**, then there is no need to include a **return** instruction

Program 4.3

```
import java.util.*;

public class FindCost5{
  public static void main(String[] args){
      Scanner sc = new Scanner(System.in);
      System.out.print("Initial price: ");
      double price = sc.nextDouble();
      System.out.print("Tax rate: ");
      double tax = sc.nextDouble();

      price = addTax(price,tax);

      System.out.println("Cost after tax = " + price);
  }
  private static double addTax(double priceIn,
                                                  double taxIn){
      return priceIn * (1 + taxIn / 100);
  }
}
```

Analysis of program 4.3

- The line that calls the method is:

```
price = addTax(price, tax);
```

- There are two items in the brackets;
  - These are the *actual* values that we are sending into our method
  - They are therefore referred to as the **actual parameters** of the method
  - Their values are copied onto the formal parameters in the called method

How does the program knows which values in the actual parameter list are copied onto which variables in the formal parameter list?

- The answer to this is that it is the **order** that is important
- In FindCost example, the value of price is copied onto priceIn; the value of tax is copied onto taxIn
- Although the variable names have been conveniently chosen, the names themselves have nothing to do with which value is copied to which variable

Using the return value

- The addTax method returns the result that we are interested in, namely the new price of the item
- We need to do is to assign this value to the variable price. As you have already seen we have done this in the same line in which we called the method:

```
price = addTax(price, tax);
```

- A method that returns a value can be used just as if it were a variable of the same type as the return value

Using the return value

```
price = addTax(price, tax);
```

- Above, we have used it in an assignment statement
- It could also, for example, be dropped into a println statement, just as if it were a simple variable of type **double**

```
System.out.println
          ("Cost after tax = " + addTax(price, tax));
```

More examples of methods

Calculate the square of a number

- We will name the method square
- It should accept a single value of type double
- It will return a double
- The single instruction will return the result of multiplying the number by itself

```
private static double square(double numberIn)
{
    return numberIn * numberIn;
}
```

We can use this method in another part of the program

- We can declare and initialize two variables as follows

```
double a = 2.5;
double b = 9.0;
```

- We want to assign the square of a to a **double** variable x and the square of b to a **double** variable y

```
x = square(a);
y = square(b);
```

- After these instructions, x would hold the value 6.25 and y the value 81.0

A method that returns the greater of two integers

- The method will be called max
- It will accept two integer values, and will return the bigger value of the two
- It will require two integer parameters
- It will return an integer

```
private static int max(int firstIn, int secondIn)
{
    if(firstIn > secondIn)
    {
        return firstIn;
    }
    else
    {
        return secondIn;
    }
}
```

A method that reports on whether or not a particular integer is an even number

- We will call the method isEven
  - It will accept a single parameter of type **int**
  - It will need to return a value of **true** if the number is even or **false** if it is not; so the return type is going to be **boolean**

```
private static boolean isEven(int numberIn)
{
   if(numberIn % 2 == 0)
   {
       return true;
   }
   else
   {
       return false;
   }
}
```

- A **boolean** method such as the one above can often be used as the test in a selection or loop

```
if(isEven(number))
{
   // code here
}
```

- To test if a number is odd

```
if(!isEven(number))
{
   // code here
}
```

Notes

- A method cannot change the **original** value of a variable that was passed to it as a parameter
- The reason for this is that all that is being passed to the method is a copy of whatever this variable contains - in other words, just a value
- The method does not have access to the original variable
- Whatever value is passed is copied to the parameter in the called method

```
public class ParameterDemo{

  public static void main(String[] args){
      int x = 10;
      demoMethod(x);
      System.out.println(x);
  }
  private static void demoMethod(int x){
      x = 25;
      System.out.println(x);
  }
}
```

What is the output?

```
public class ParameterDemo{

  public static void main(String[] args){
      int x = 10;
      demoMethod(x);
      System.out.println(x);
  }
  private static void demoMethod(int x){
      x = 25;
      System.out.println(x);
  }
}
```

```
25
10
```

Variable scope

- Variables are only "visible" within the pair of curly brackets in which they have been declared
  - If they are referred to in a part of the program outside these brackets, then you will get a compiler error
- Variables that have been declared inside the brackets of a particular method are called **local** variables
  - Variables price and tax are said to be local to the main method
- We say the variables have a **scope**
  - Their visibility is limited to a particular part of the program
  - If price or tax were referred to in the addTax method, they would be out of scope

Program 4.5

```
public class ScopeTest
{
   public static void main(String[] args)
   {
      int x = 1;      // x is local to main
      int y = 2;      // y is local to main
      method1(x, y); // call method1
   }

   private static void method1(int xIn, int yIn)
   {
      int z;          // z is local to method1
      z = xIn + yIn;
      System.out.print(z);
   }
}
```

Program 4.6

```
// this program will give rise to two compiler errors
public class ScopeTest2
{
   public static void main(String[] args)
   {
      int x = 1;       // x is local to main
      int y = 2;       // y is local to main
      method1(x, y); // call method1
      System.out.print(z); /* this line will cause a compiler
                               error as z is local to method1 */
   }

   private static void method1(int xIn, int yIn)
   {
      int z;           // z is local to method1
      z = x + y;       /* this line will cause a compiler error as
                          x and y are local to main */
      System.out.print(z);
   }
}
```

Summary

- A method can access variables that have been declared as formal parameters
- A method can access variables that have been declared locally - in other words that have been declared within the curly brackets of the method
- As you will learn in chapter 7, a method has access to variables declared as *attributes of the class*
- A method cannot access any other variables

Method overloading

- You have already encountered the term *overloading* in previous chapters, in connection with operators. For example,
  - The division operator (/) can be used for two distinct purposes:
    - division of integers
    - division of real numbers
  - The + operator can be used for:
    - addition
    - concatenating two strings
- So the same operator can behave differently depending on what it is operating on - operators can be overloaded
- Methods too can be overloaded

Consider the max method again

```
private static int max(int firstIn, int secondIn)
{
   if(firstIn > secondIn)
   {
      return firstIn;
   }
   else
   {
      return secondIn;
   }
}
```

- The max method accepts two integers and returns the greater of the two
- What if we wanted to find the greatest of three integers?

- What if we wanted to find the greatest of three integers?
- We could write a new method with the following header

```
private static int max(int firstIn, int secondIn, int thirdIn)
{
    // code goes here
}
```

- Both methods have the same name but the parameter list is different; each one will *behave* differently
  - We have given this method the same name as before
  - This time it has *three* parameters instead of two
- We can declare and call both methods within the same class

Polymorphism

- When two or more methods, distinguished by their parameter lists, have the same name but perform different functions we say that they are **overloaded**
- Method overloading is one example of what is known as **polymorphism**
- Polymorphism literally means *having many forms*, and it is an important feature of object-oriented programming languages
- It refers to the phenomenon of having methods and operators with the same name performing different functions

Polymorphism

How, when we call an overloaded method, does the program knows which one we mean?

- It is determined by the actual parameters that accompany the method call
- They are matched with the formal parameter list, and the appropriate method will be called.

```
public class findMax{
  public static void main(String[] args){
      System.out.println("max: " + max(10, 56));
      System.out.println("max: " + max(2, 6, 9));
  }
  private static int max(int first,int second){
      if (first > second)
            return first;
      else
            return second;
  }
  private static int max(int first,int second,int third){
      int result = first;
      if (second > result)
            result = second;
      if (third > result)
            result = third;
      return result;
  }
}
```

An alternative way to implement the second version of max

- We could start off by finding the maximum of the first two integers, using the first version of max
- We could then do the same thing again, comparing the result of this with the third number

```
private static int max(int firstIn, int secondIn, int thirdIn)
{
   int step1, result;
   step1 = max(firstIn, secondIn); // call the first version of max
   result = max(step1, thirdIn);   // call the first version of max again
   return result;
}
```

Consider program 4.8

- The program allows a user to process the sale of tickets for some event.
- Four menu options are offered:
  - Option 1: Displays the total amount of money required to purchase a number of tickets; children are charged half-price
  - Option 2: Display the total amount of money required to purchase a number of tickets; the total cost is added to the grand total
  - Option 3: Display the grand total
  - Option 4: Terminate the program
- You can see that we have had to declare a new Scanner object in each method - now that you understand the notion of variable *scope*, you should understand why we have had to do this

Program 4.8

```
import java.util.*;

public class TicketVendor{

   public static void main(String[] args){
      final double PRICE = 30; // price of an adult ticket
      double total = 0, cost;
      char choice;
      do{
         choice = displayMenu();
         switch (choice){
            case '1' : calculateCost(PRICE);
                       break;
            case '2' : cost = calculateCost(PRICE);
                       total += cost;
                       break;
            case '3' : displayTotal(total);
                       break;
            case '4' : break;
            default  : System.out.println("Invalid choice");
         }
      } while (choice != '4');
   }
```

Program 4.8

```
private static char displayMenu(){
   Scanner sc = new Scanner(System.in);
   System.out.println("1. Get cost of tickets");
   System.out.println("2. Record purchase of tickets");
   System.out.println("3. View total received so far");
   System.out.println("4. Quit");
   System.out.println();
   System.out.println("Enter a number from 1 - 4");
   return sc.next().charAt(0);
}
```

Program 4.8

```
   // calculates and displays the cost of tickets
   private static double calculateCost(double priceIn){
      Scanner sc = new Scanner(System.in);
      int adult, child;
      double cost;

      System.out.print("How many adult tickets required? ");
      adult = sc.nextInt();
      System.out.print("How many child tickets required? ");
      child = sc.nextInt();
      cost = (adult + 0.5 * child) * priceIn;
      System.out.println("Total cost will be " + cost);
      return cost;
   }

   // displays the current total
   private static void displayTotal(double totalIn){
      System.out.println("Total received " + totalIn);
   }
}
```