5 - METHODS

5.1 What are methods?

A method is a collection of instructions put together under a single *signature*. A signature is a name followed by optional parameters. We cover parameters in the next section. Once a method has been *defined,* a programmer can *call the method* by invoking its signature anywhere in a program. A method definition consists of its signature and its instructions. The instructions are typically Java commands and/or other method calls.

In our Java programs we have already made use of methods and we have already invoked their signatures. For example, when we want to output a message to the screen we use the statement

```
Stdout.println("Hi, my name is Albert");
```

This statement contains a call to the the method `println(...)`. This method has been *predefined* by programmers so that we do not need to worry about its inner instructions. These inner instructions are responsible for making sure that the message `"Hi, my name is Albert"` is sent to and printed on the screen. We do not need to know what the method does. All we need to know is how to use it.

Java, like all programming languages, allows us to *define our own methods*. When we define our own methods we are *extending the language* to contain our own methods. In other programming languages, methods may be called *procedures* or *functions*. For example, in C and C++ methods are called *functions*. In Pascal and Turing, methods are called *procedures* and *functions*.

When we define our own method, we can use it and reuse it as many times as we wish without having to rewrite its inner instructions, and other programmers can make use of our methods by simply knowing how to use them without ever knowing about its inner workings. Here is an example of a method that prints an address:

```
     public static void address() {
          Stdout.println("Peter Jones");
          Stdout.println("100 Greenwood Avenue");
          Stdout.println("Toronto, Ontario");
          Stdout.println("M4J 4B7");
          Stdout.println("---------------------");
     }
```

To make use of this method all we need to do is place in our program a *call* to it. This call will invoke the method and execute *all the instructions of the method.* Then program control will be returned to the point where the call was made. The following contains the full program with the method definition, and a call to it:

```
import hsa.*;
public class Learning {
     public static void main(String[] args) {
          address();
     }

     public static void address() {
          Stdout.println("Peter Jones");
          Stdout.println("100 Greenwood Avenue");
          Stdout.println("Toronto, Ontario");
          Stdout.println("M4J 4B7");
          Stdout.println("---------------------");
     }
}
```

Important things to note:

- The method `address()` is defined *outside* the `main()` method
- The method `address()` may be defined *before* or *after* the `main()` method.
- The method signature `address()` begins with a lower case letter
- The method `address()` is preceded by the keywords `public static void`. Here is a quick explanation of these:
- The method is `public` in that it is accessible from other methods. We will learn also about `private` and `protected` methods.
- The method is `static.` This means that the method is not associated with any particular object.

- The method is *declared* as `void.` This means that the method *does not return any value*. Methods can return any types. We will do an example below of a method that returns an integer.

5.1 Exercises

1. Copy, compile, and run the class `Learning` explained above.

2. Modify the program of exercise 1. Create another method called `contactInfo()` that prints out a phone number, a fax number, an e-mail address, and a web site address. Then place the call to this new method in the appropriate place inside the `main()` method so that the output shows both the address followed by the contact information.

3. Create a method that displays a menu of 4 options as follows:

```
Analytic Geometry Menu:

1. Midpoint between two points
2. Distance between two points
3. Equation of a line
4. Quit

Enter an option:
```

   The method returns the choice selected by the user as an integer. Once the value has been returned, the main method prints a message like

```
You have selected option 3
```

   and then re-displays the menu. If the user enters something other than an option from 1 to 4, an appropriate error message is displayed. The method requires a definition as follows:

```
public static int analyticGeometryMenu() {
     int choice = -1;

     ... <statements> ...

     return choice;
}
```

5.2 Methods and Parameters

Some methods of the previous section do not return a value, i.e., they are declared as *void*. Some others return a value of type integer, and thus declared as *int*. Besides returning a value, methods can also *take in* values.

We do this through the use of *parameters*, sometimes also called *arguments*. Here is a sample declaration of a method that returns an integer type and takes in two parameters also of integer type:

```
     public static int maximum(int m, int n) {
          if (m > n)
               return m;
          else
               return n;
     }
```

The method `maximum(int m, int n)` takes two integers which are then compared inside the method. The larger of the two numbers is returned. Here is an example on how to use it:

```
import hsa.*;
public class MyFirstMethodWithParameters {
     public static void main(String[] args) {
          int x = -1;
          int y = -1;

          Stdout.print("Enter first number: ");
          x = Stdin.readInt();

          Stdout.print("Enter second number: ");
          y = Stdin.readInt();

          int larger = maximum(x, y);
          Stdout.println("The larger number is " + larger);
     }

     public static int maximum(int m, int n) {
          if (m > n)
               return m;
          else
               return n;
     }
}
```

Important things to note:

- The method `maximum(int m, int n)` is defined *outside* the `main()` method.
- The *order of the parameters in the definition of*

```
maximum(int m, int n)
```

is very important. When the call is placed to the method, the parameters in the call `larger = maximum(x, y)` correspond directly to the parameters in the definition of `maximum(int m, int n)`

The parameters in the method definition are called *formal parameters*. The values passed in the call are called *actual parameters.*

In this particular case, when a call to a method is placed, a *copy* of the actual parameters is made to the formal parameters. This is known as *passing by value*. The parameters inside the method have a different memory location. This means that any changes we make to the parameters inside the method *do not affect any variables* of the calling method. For example:

```
import hsa.*;
public class MySecondMethodWithParameters {
     public static void main(String[] args) {
          int x = 0;

          Stdout.print("Value of x: ");
          Stdout.println(x);
          printMoreX(x);
          Stdout.print("Value of x: ");
          Stdout.println(x);
     }

     public static void printMoreX(int x) {
          x = x + 1;
          Stdout.print("Value of x: ");
          Stdout.println(x);
          x = x + 1;
          Stdout.print("Value of x: ");
          Stdout.println(x);
          x = x + 1;
          Stdout.print("Value of x: ");
          Stdout.println(x);
```

```
            }
    }
```

The output of this program is:

```
Value of x: 0
Value of x: 1
Value of x: 2
Value of x: 3
Value of x: 0
```

Because the formal parameters are copies of the actual parameters, the names of the parameters are kept separately in the computer's memory. This allows us to have formal parameters that have the same name as the actual parameters. They are kept in memory as separate locations.

We will learn another form of parameter passing knows as *passing by reference*. The behaviour of the parameters is different in this case.

Case Study - Text File Input/Output

In this case study we learn how text files are created and how information is stored in them. We will both write and read information to and from files. In general, text files under any operating system have these characteristics:

- They contain ASCII readable characters
- They can be opened with a simple text editor such as TextEdit, Notepad, or the editors of IDE's.
- They do not contain any formatting such as fonts, text sizes, colours, or graphics
- Source code programs are stored as text files so they can be opened with simple text editors. All our Java code is stored in text files.
- Text files usually (but not always) have the suffix `txt` attached at the end of the name of the file. This suffix is called the *file extension.* Here are some file extensions for source code. Files with these extensions will be text files:

Extension	Language	Example
java	Java	hello.java

c	C/C++/C#	sudoku.c
pas	Pascal	inventory.pas
t	Turing	circle.t

We may use any extension we want for text files, but `txt` is always a good idea as it is recognized across all operating system platforms.

The sequence of steps when using files is:

1. Open file
2. Perform read or write operations on the file
3. Close file

1.Opening a file

The following Java code will successfully open a text file in read/write mode. This means that, once the file is open, we can both read the contents of the file into memory variables and/or write the contents of memory variables into the file. If the file does not exist, this code will create a new file with a size of zero bytes.

```
import java.io.*;
public class FileOperations {
   public static void createFile(String fileName) {
        RandomAccessFile f;

        try {
             f = new RandomAccessFile(fileName, "rw");
        }
        catch (IOException e) {
             System.out.println(e);
        }
   }
   public static void main(String[] args) {
        FileOperations.createFile("names.txt");
   }
}
```

- The statement `import java.io.*` is necessary for file operations

- The code declares `f` as a `RandomAccessFile` which will allow us to access the file from our own code.

- In this program, the file `names.txt` is opened for reading and writing with `"rw"`. If the file does not exist, it is created empty. If we try to open it with a text editor the file will contain nothing.

- The actual opening/creating of the file needs to happen inside the `try-catch` construct. This is necessary because
  - Java is not performing the actual file operations. It is the underlying operating system (Windows, Unix, Linux, OS-X) that does the actual file work.
  - The Operating System returns useful error codes to our programs that we can report to the user if necessary.
  - We can prevent programs from crashing and give the user meaningful error messages such as `File not found` or `Disk not available` or `Insufficient user rights`

- Some of the most common file operation errors happen in our Java programs when we try to

  - Open a file for reading but the file does not exist (read error)
  - Create a file but the disk is full (write error)
  - Create a file but we don't have permissions in the folder where we want to create the file (write error)
  - Overwrite a file that is read-only (write error)

- The `try-catch` construct cannot
  - Catch programming errors such as reading beyond the end of the file. Maintaining read operations within the file is similar to keeping a string pointer within bounds. It is the programmer's responsibility. The error in the statement

    ```
    System.out.println("hello".charAt(10))
    ```

    will not be caught by `try-catch`

2. Read/Write Operations

Once a file has been opened/created, Java allows us to perform read/ write operations to the file. The commands to perform these operations must be inside the `try-catch` construct. For example, if `f` is an open text file, then the code

```
try {
   f.writeBytes("Jonathan");
}
catch (IOException e) {
   System.out.println(“Error writing to file”);
}
```

will write the word `Jonathan` into the file.

3. Closing a file

It is necessary to always close a file. The write operations that we have in a program *do not guarantee that the information is actually on file*. The operating system ultimately makes the decision as to when to write information to disk. This is so that the computer can perform optimally.

Closing a file is an *explicit command to the operating system* to *flush* the data from memory to the disk. This is directly related to the ejecting of USB keys. When a USB key is pulled out without ejecting, the files have been left open. When a key is ejected all files have been closed.

Example - Create a file and write to it
The following code creates a file called `countries.txt`, writes a few entries and then closes the file. Once the program has run, we can open the file with an editor to see its contents:

```
import java.io.*;
public class Geography {
    public static void main(String[] args) {
        RandomAccessFile countries;
        try {
            countries = new RandomAccessFile("countries.txt", "rw");
            countries.writeBytes("Canada\n");
            countries.writeBytes("France\n");
            countries.writeBytes("Mexico\n");
            countries.close();
        }
        catch(IOException e) {
            System.out.println("Error opening file");
        }
    }
}
```

Example - Read from a file and print contents on screen
The following code opens the file `countries.txt`, reads one line and prints it to the screen. It then continues reading and printing lines of the file until the end of the file is reached

```
import java.io.*;
public class ShowCountries{
    public static void main(String[] args) {
        RandomAccessFile countries;
        try {
            countries = new RandomAccessFile("countries.txt", "r");
            String line = countries.readLine();
            while (line != null) {
                System.out.println(line);
                line = countries.readLine();
            }
            countries.close();
        }
        catch(IOException e) {
            System.out.println(e);  // use e for error message
        }
    }
```

5.2 Exercises: Methods with parameters

1. Copy, compile, and run the classes `MyFirstMethodWithParameters` and `MySecondMethodWithParameters` given above

2. Write a Java method with the signature

   ```
   public static boolean multipleFactors(int p, int q, int r)
   ```

   that returns `true` if `p` is a factor of `q` and `q` is a factor of `r`. This method can be written with only one statement inside the method.

3. Write a Java method with the signature

   ```
   public static boolean xor(boolean p, boolean q)
   ```

   that returns the value of `p XOR q`
4. Use the method `xor()` of the previous question to write another method that returns the value of `p XNOR q`

5. Write a Java method with the signature

```
public static double distanceBetweenTwoPoints(int x1,
                                              int y1,
                                              int x2,
                                              int y2)
```

where `(x1, y1)` represents a point and `(x2, y2)` represents another point on the plane. The method returns the distance between the two points. Insert a call to this method in the menu that we created for exercise 10.1 3, option 2 so that the method is invoked when the user choose option 2 of the menu.

6. Write the methods with the necessary parameters to complete the options 1 and 3 of the menu of exercise 10.1 3. Two methods are needed here: One to calculate the midpoint between two points and another to determine the equation of a line from two points.

7. Write a Java method with the signature

```
public static String derivative(String monomial)
```

that returns the derivative of the given monomial. Assume that the monomial has the form

ax^n

where *a* is the coefficient, *x* is the independent variable and *n* is the exponent. The derivative of a monomial is defined as

nax^(n - 1)

For example, a call like

```
System.out.println(derivative("-4x^3"))
```

prints out

```
-12x^2
```

8. Write a method with the signature:

```
public static String monthStr(int month)
```

that returns the name of the month from the given month number, where 1 is January, 2 is February, etc.

9. Write a method with the signature:

```
public static String generateCode(int size)
```

that returns a `String`, containing as many digits as specified in `size`. Each digit is a random number in the range [0, 9]. For example,

```
generateCode(9)
```

returns a string of 9 random digits like `443523872`. To generate a random integer in the range [0, 10) with Java, use the command

```
(int) (Math.random() * 10)
```

10. Write a Java method with the signature

```
public static String reversal(String message)
```

that takes a message and reverses it. The method does not print the message. It just returns the reversed message.

11. Write a Java method with the signature

```
public static boolean isPalindrome(String message)
```

that takes makes use of the method `reversal()` to determine if `message` is a palindrome.

12. Write a Java method with the signature

```
public static String removeSpaces(String message)
```

that returns `message` with all its spaces removed.

13. Books are coded using the International Standard Book Number: ISBN. There are two systems: ISBN-10 and ISBN-13

ISBN-10

ISBN 0-205-02880-2

9 780205 028801

ISBN-13

ISBN 979-345678901-8

9 793456 789018

In this exercise we determine whether a given book code is a valid ISBN-10 number. A valid ISBN-10 must satisfy all the following conditions:

- Code is exactly 10 characters long
- The last digit of the code is in the range 0 to 10, where 10 is represented with the letter X
- All other digits are in the range 0 to 9
- A number is a valid ISBN-10 number if the following calculation is satisfied

$$\left[\sum_{i=0}^{8}(i+1)x_i\right] \bmod 11 = x_9$$

which equals

$$(x_0 + 2x_1 + 3x_2 + 4x_3 + 5x_4 + 6x_5 + 7x_6 + 8x_7 + 9x_8) \bmod 11 = x_9$$

where the leftmost position starts at zero and each subscripted variable stands for a digit in the given code. Write a Java method with the signature

```
public static boolean isbn10(String code)
```

that determines whether the code passed in the parameter is a valid ISBN-10 number. The method *does not print anything*. Any printing must be done from the calling method, not from the method `isbn10()`. Use the following codes to test that the method works correctly:

call	result
`isbn10("842199820X")`	TRUE
`isbn10("0921598424")`	TRUE
`isbn10("44637281")`	FALSE
`isbn10("56307X2448")`	FALSE
`isbn10("8421998203")`	FALSE

14. We continue the study of ISBN book codes in this exercise. The conditions for a valid ISBN-13 code are,

- Code is exactly 13 characters long
- The last digit of the code is in the range 0 to 10, where 10 is represented with the letter X
- All other digits are in the range 0 to 9
- A number is a valid ISBN-13 number if the following calculation is satisfied

$$10 - \left[\left[\sum_{i=0}^{5} (x_{2i} + 3x_{2i+1}) \right] \bmod 10 \right] = x_{12}$$

where the leftmost position starts at zero and each subscripted variable stands for a digit in the given code.

a) Expand the previous sum expression.

b) Write a Java method with the signature

```
public static boolean isbn13(String code)
```

that determines whether the code passed in the parameter is a valid ISBN-13 number. Use the following codes to test the method,

call	result
`isbn13("9780684856094")`	`TRUE`
`isbn13("9780306406157")`	`TRUE`
`isbn13("113224331877")`	`FALSE`
`isbn13("2641X71424488")`	`FALSE`
`isbn13("9780684856095")`	`FALSE`

15. Write a Java method with the signature

```
public static boolean isDigit(String s)
```

that returns `true` if every character of the string `s` is a digit (any number between 0 and 9). If not, then the method returns `false`.

16. Write a Java method with the signature

```
public static String simplify(String fraction)
```

that takes a fraction in the form of a string. The method returns a fully simplified fraction. If the fraction has a zero in the denominator the method returns the word `undefined`. In order to write this method we need to calculate the greatest common divisor of two numbers. A helping method is given below: `euclid()`

To test the `simplify()` method we use the following calls:

call	result
`simplify("10/7")`	`10/7`
`simplify("24/10")`	`12/5`

`simplify("-24/10")`	-12/5
`simplify("24/-10")`	-12/5
`simplify("-24/-10")`	12/5
`simplify("24/8")`	3
`simplify("-8/-1")`	8
`simplify("6/0")`	undefined

The following method returns the greatest common divisor of two positive integers. It is an implementation of Euclid's *greatest common divisor algorithm*:

```
private static int euclid(int m, int n) {
    while (m != 0 && n != 0) {
        if (m > n)
            m = m - n;
        else
            n = n - m;
    }

    if (m != 0)
        return m;
    else
        return n;
}
```

17.Write a Java method with the signature

```
public static int card()
```

that has no parameters. The method generates and returns an integer from 1 to 52 where each integer stands for a card in a deck of 52 cards. To generate a random integer, look at exercise 9. It shows how to generate an integer from 0 to 9. Make adjustments to the random method so that a number from 1 to 52 is generated instead.

18.Write a Java method with the signature

```
          public static long factorial(int n)
```

that calculates the factorial of the number `n`. Factorial is defined as the multiplication of `n` with every integer smaller than `n` down to 1. Factorial of a number `n` is written `n!` For example,

```
4! = 4 x 3 x 2 x 1 = 24
6! = 6 x 5 x 4 x 3 x 2 x 1 = 720
```

Factorials get very large very quickly. The value of `n` cannot be very large because the calculation will not be possible even if the method returns a `long`. It would be necessary to use scientific notation to express the very big numbers.

19.Write a Java method with the signature

```
public static String phoneWithoutDashes(String phoneNumber)
```

that returns `phoneNumber` with the dashes removed. For example,

`phoneWithoutDashes("416-833-2622")` returns `4168332622`

20.Write a Java method with the signature

```
public static int quadraticSolutions(int a, int b, int c)
```

that returns the number of solutions of the quadratic equation $f(x) = ax^2 + bx + c$. For example, if $f(x) = -2x^2 + x + 5$ the method returns 2 since $f(x)$ has two *x*-intercepts.

21.Write a Java program that asks the user for a file name. Then the program creates a file with the given file name. The structure of the program is

```
import the needed classes
public static void main(String args[]) {
   String filename = ask user for a file name
   declare f as a RandomAccessFile and initialize to null
   try {
        f = open the file with the given filename
        close the file
   }
   catch(IOException e) {
        print an error message
   }
}
```

22. Write a Java program that creates a text file called `log.txt`, asks the user for a phone number writes it on the file, then closes the file. The structure of the program is

```
import the needed classes
public static void main(String args[]) {
   declare the needed variables
   try {
        create the file with name "log.txt"
        get a phone number from the user
        write the number in the file "log.txt"
        close the file
   }
   catch(IOException e) {
        print error message
   }
}
```

23. Write a Java program that asks the user for the name of a text file and prints its contents on the system window. The program structure is,

```
import the needed classes
public static void main(String args[]) {
   String filename = ask user for a file name
   declare f as a RandomAccessFile and initialize to null

   try {
        f = open the file with the given filename

        String line = read a line from the file
        while the end of file has not been reached {
             print line
             line = read a line from the file
        }

        close the file
   }
   catch(IOException e) {
        print an error message
   }
}
```

24. Write a Java program that

a) Creates a file called `records.txt`
b) Asks the user for a message and writes the message in the file as soon as it is entered
c) The program keeps asking for a message and keeps on writing each message every time one is entered. Remember to include a `\n` in your `writeBytes()` call
d) When the message `quit` is entered by the user, the program ends but does not write the word `quit` to the file

25. Write a Java method with the signature

```
public static void copyFile(String source, String target)
```

that copies all the contents of the file `source` to the file `target`. The algorithm for the method is

```
open source file
if source file opens successfully then
     open target file
     if target file opens successfully then
          line = read a line from source file
          while end of source file has not been reached {
               write the line to target file
               line = read next line from source file
          }
          close target file
          close source file
     }
     else
          print error: target file could not be opened
          close source file
     end if
else
     error: source file could not be opened
end if
```

26. Write a Java method with the signature

```
public static void encryptFile(String source)
```

that encrypts the file given by `source`.

We encrypt a file by making a copy of `source` in the same way we did in exercise 25, but before writing a line of bytes to the target file we encrypt the line, then write it. For a target filename, we make up a name. For example, if the source filename is `assignment.txt` we can make up a name like `assignment.enc`.

The algorithm is (with encryption statements in purple),

```
open source file
```

```
    if source file opens successfully then
         **make up a target filename**
         open target file with target filename
         if target file opens successfully then
              line = read a line from source file
              while end of source file has not been reached {
                   **encrypt line**
                   write the encrypted line to target file
                   line = read next line from source file
              }
              close target file
              close source file
         else
              print error: target file could not be opened
              close source
         end if
    else
         error: source file could not be opened
    end if
```

27.Write a Java method with the signature

```
public static void compareFiles(String fName1, String fName2)
```

that determines if the files `fName1` and `fName2` are identical. As soon as a difference is found, the method reports where the difference has been found and stops comparing immediately. Write this method without breaking or returning from inside a loop. The algorithm is,

```
open files
if both files are the same size
   line1 = read from f1
   line2 = read from f2
   same = line1.equals(line2)
   while not f1.eof() and not f2.eof() and same {
        line1 = read from f1
        line2 = read from f2
        same = line1.equals(line2)
   }
else
   report error: files are different in size
end if
close files
```