Software and Programming I

Lab 2:
Step-by-step execution of programs using a Debugger

Tobi Brodie (tobi@dcs.bbk.ac.uk)

Lab Session 2: Objectives

This session we are concentrating on BlueJ’s built-in debugging tool.

In order to fully understand how our programs work we use the debugging tool to see the values of variables at different points throughout execution.

We do this by adding breakpoints into our code.

Breakpoints will halt the code execution when we run it, allowing us to inspect the values held in variables in the currently executing methods.

Lab Session 2: Exercises

There are 4 exercises in this week's lab presentation. You will be taken step-by-step through Exercise 1, modifying the InterestCalculator class created last week and then using the debugging tool to understand its execution.

Exercises 2 & 3 are more complex exercises, which you may be able to complete within the lab session.

Marked Exercise 1 is the first of the six exercises that count towards your **coursework**, and no assistance will be given to complete it. The exercise **must** be completed by 6th February. You will be asked to show it and explain its execution.

In–Lab Marked Exercises

There are 6 exercises that count for 5% towards your final module mark. Typically you will have 2 weeks to complete each exercise; however, it is recommended that you complete them within a week, in order to check that your solution is correct.

Your marked exercise will be reviewed and assessed in the lab by one of the lab assistants.

If, for any reason you cannot complete the work before the deadline, you will need to apply directly to the university for consideration of mitigating circumstances. The MIT-CIRCS form is available through the MyBirkbeck portal. As the exercises count towards your final mark, **lab assistants are unable to give extensions.**

In–Lab Marked Exercises (2)

Birkbeck UNIVERSITY OF LONDON

The assistant will require you to explain your code and may change the code and then ask you to explain how those changes affect the program. Once the lab assistant is happy that you fully understand the work, they will record the exercise as completed. If further work is required, this will also be recorded.

Marks will be uploaded to Moodle before the next session. It is **your responsibility** to check that the marks have been recorded correctly. To ensure that any issues are resolved quickly, you should always take note of the lab assistant's name and the date that your work was seen.

Failure to complete the **6 Marked Exercises** within their two-week deadlines will **disqualify** you from the second in-class test, which in itself counts for 10% of your final mark.

Exercise 1: InterestCalculator2

- Launch BlueJ - begin with the **Start** icon in the lower left corner of the screen. Select the options

  Start -> **All Programs** -> **Departmental Software** -> **Computer Science** -> **BlueJ**

  or start typing **BlueJ** in the box – the app icon should appear
- Create a new Project on your disk space.
  1. Select Project then followed by **New Project**.
  2. Select a directory in your disk space and a suitable name for your project, e.g. **week2**. After entering **week2** in the BlueJ window, a new BlueJ window will appear for the project **week2**.
- Create a new class by clicking on button **New Class ...** in the new BlueJ window. Enter the name **InterestCalculator2** for the new class and click on **OK**.

Exercise 1: InterestCalculator2 (2)

We will be using the code used in last week's InterestCalculator class as the starting point for this week's first exercise. Copy the code completed last week into InterestCalculator2, changing the class name by adding 2 to it.

```
public class InterestCalculator2 {
          public static void main(String[] args)
          { …
```

InterestCalculator can be found at:
http://www.dcs.bbk.ac.uk/~roman/sp1/java/InterestCalculator.java

Exercise 1: InterestCalculator2 (3)

We will then add code to the class that allows the user to set a saving target and then informs the user when this target has been met.

To do this we use the `Scanner` class to receive a value from the user, and add a new variable of type **`double`** to store the savings target.

We must also import the Java utility `Scanner` class into our program by including the following line at the beginning of our code, before the class declaration:

```
import java.util.Scanner;
```

Exercise 1: InterestCalculator2 (4)

We then use the following code within the main method to get the value from the user. First, we use `System.out.println` to open the terminal window showing an instruction to the user *(the terminal needs to be open for user input)*

```
System.out.println("Please enter a savings target");
```

We then write the following code to create a `Scanner` class instance, `scan`, to use to get the input:

```
Scanner scan = new Scanner(System.in);
```

Finally, we store the value in a variable of type **`double`**:

```
double savingsTarget = scan.nextDouble();
```

Structure of InterestCalculator2

The program should now look like this:

```
import java.util.Scanner;

public class InterestCalculator2
{
  public static void main(String[] args)
  {
    double initialBalance = 10000;
    System.out.println("Please enter " +
                       "a savings target");
    Scanner scan = new Scanner(System.in);
    double savingsTarget = scan.nextDouble();
    System.out.println("The initial balance is: £" +
                        initialBalance);
  …
```

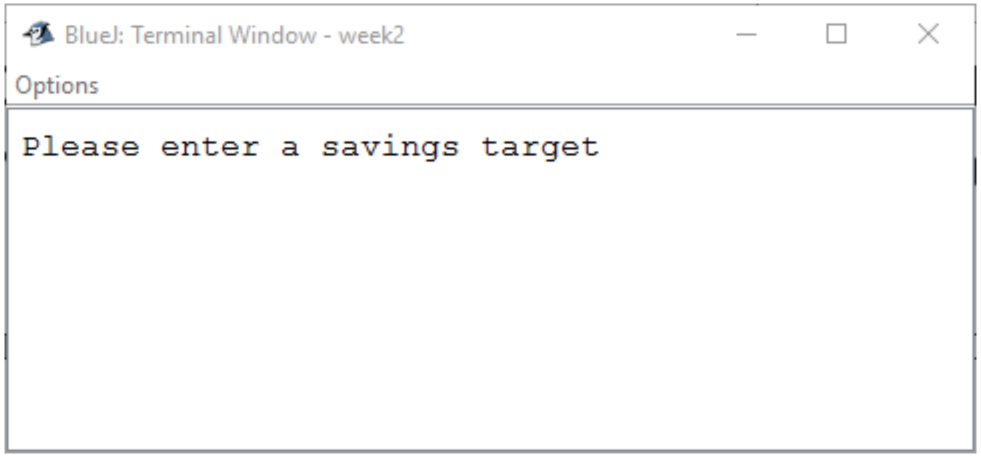
Exercise 1: InterestCalculator2 (5)

Now that a savings target value has been set, an **`if`** statement should be introduced into the program to check the current balance against this value to see if the target has been reached. The following code will need to be inserted into our program each time the current balance is updated:

```
if (currentBalance > savingsTarget)
{
      System.out.println("Congratulations, your " +
                 "savings target has been reached");
}
```

Exercise 1: InterestCalculator2 (6)

After compiling the code and executing the method the terminal window should display the following:

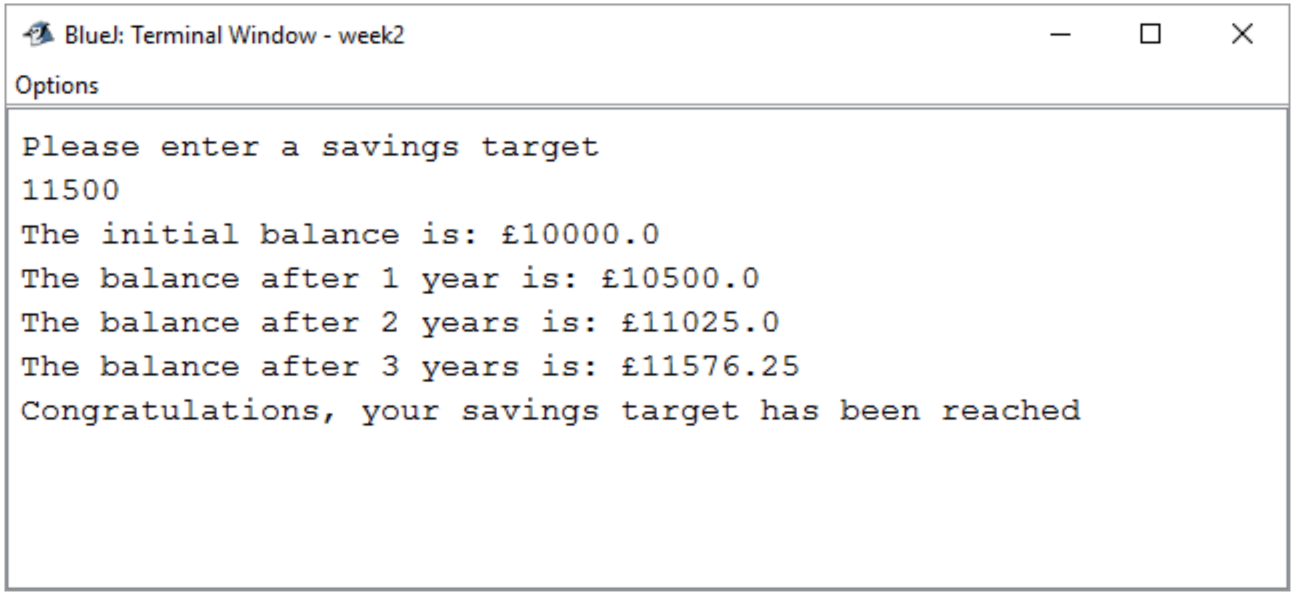


```
Please enter a savings target
```

Enter a value of 11500 as the savings target.

Exercise 1: InterestCalculator2 (7)

The terminal window output should be similar to the screenshot below:

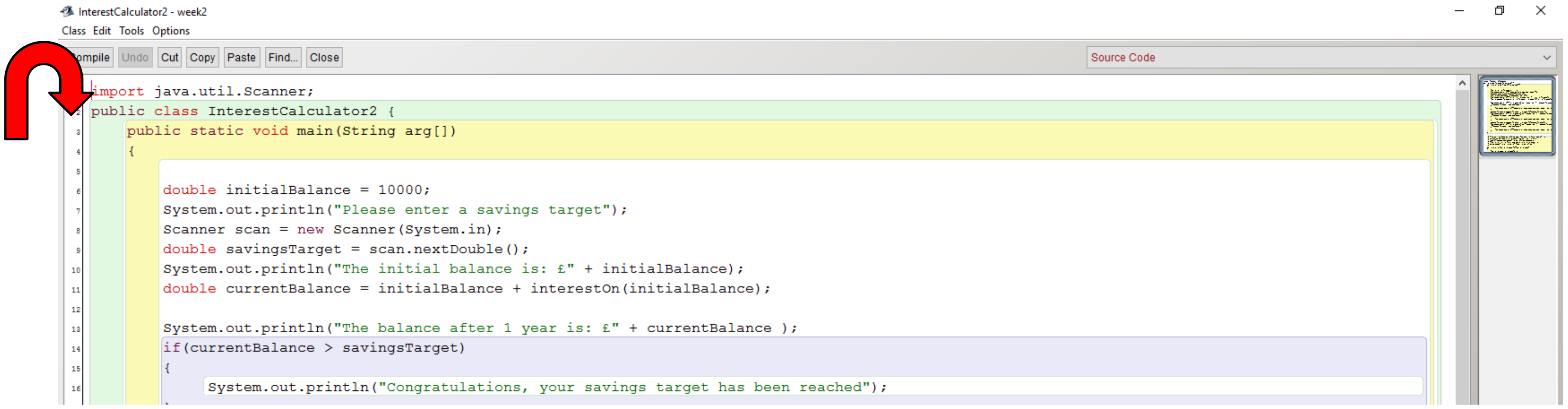


```
Please enter a savings target
11500
The initial balance is: £10000.0
The balance after 1 year is: £10500.0
The balance after 2 years is: £11025.0
The balance after 3 years is: £11576.25
Congratulations, your savings target has been reached
```

Test the program again using the target of £11025. You will see the program is not working as expected. We should now debug the program to find the issue.

Debugging InterestCalculator2

- By utilising the built-in Debugger in BlueJ we can make sure that the values of the variables are as expected and that our program will print the “congratulations” message at the correct point.
- Before you can start debugging your code you must compile it.
- Once compiled the section to the left of your code will turn white (see figure below).


InterestCalculator2 - week2
Class Edit Tools Options
ompile Undo Cut Copy Paste Find... Close
Source Code

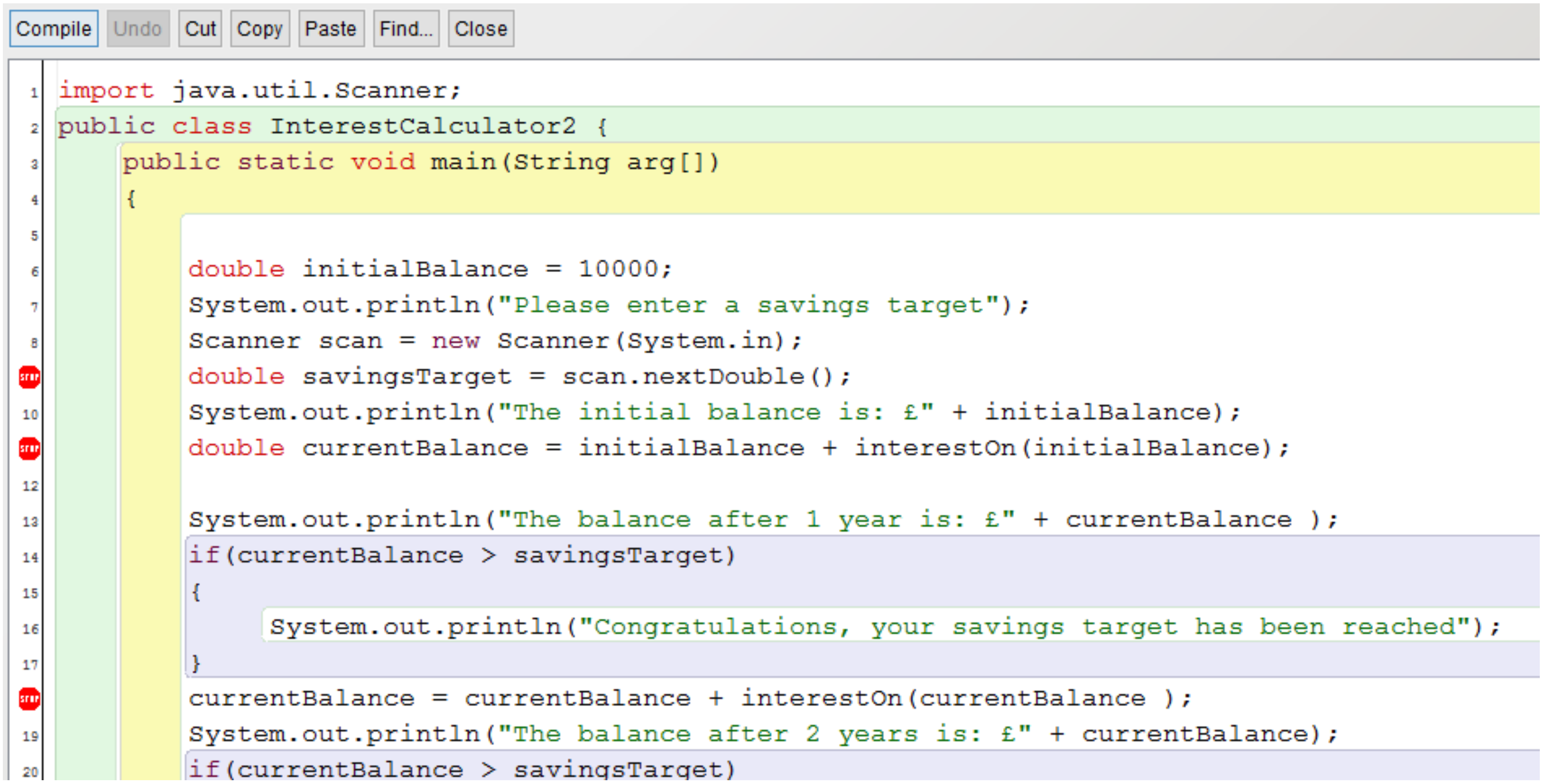
```
import java.util.Scanner;
public class InterestCalculator2 {
    public static void main(String arg[])
    {

        double initialBalance = 10000;
        System.out.println("Please enter a savings target");
        Scanner scan = new Scanner(System.in);
        double savingsTarget = scan.nextDouble();
        System.out.println("The initial balance is: £" + initialBalance);
        double currentBalance = initialBalance + interestOn(initialBalance);

        System.out.println("The balance after 1 year is: £" + currentBalance );
        if(currentBalance > savingsTarget)
        {
            System.out.println("Congratulations, your savings target has been reached");
```

Debugging InterestCalculator2 (2)

Breakpoints can then be added into our code by clicking within the white section. Breakpoints will halt the code execution when we run it, allowing us to inspect the values held in variables in the currently executing methods.



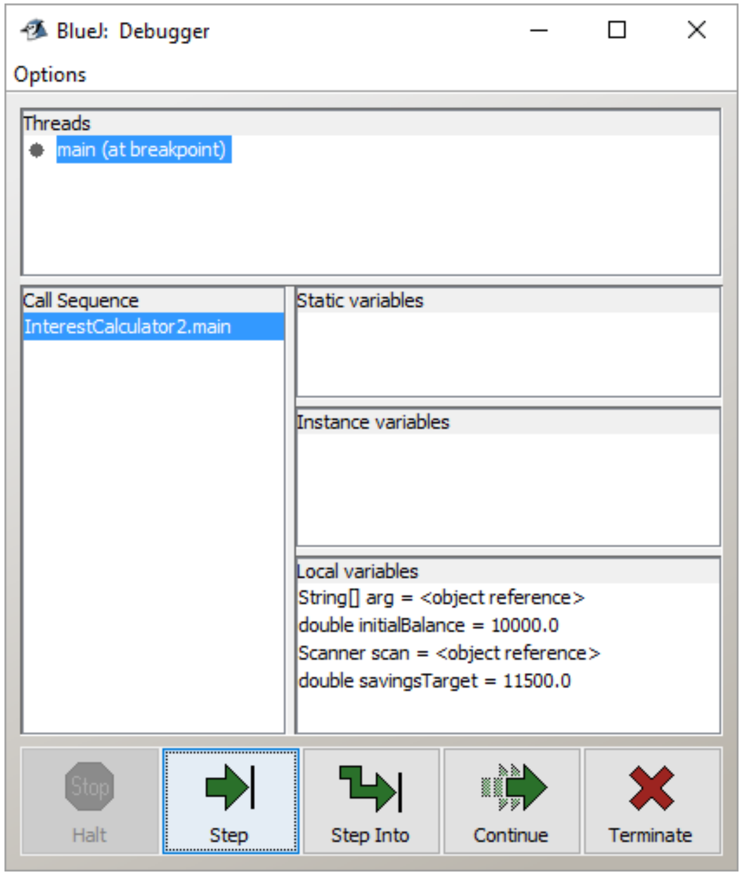
```
import java.util.Scanner;
public class InterestCalculator2 {
    public static void main(String arg[])
    {

        double initialBalance = 10000;
        System.out.println("Please enter a savings target");
        Scanner scan = new Scanner(System.in);
        double savingsTarget = scan.nextDouble();
        System.out.println("The initial balance is: £" + initialBalance);
        double currentBalance = initialBalance + interestOn(initialBalance);

        System.out.println("The balance after 1 year is: £" + currentBalance );
        if(currentBalance > savingsTarget)
        {
            System.out.println("Congratulations, your savings target has been reached");
        }
        currentBalance = currentBalance + interestOn(currentBalance );
        System.out.println("The balance after 2 years is: £" + currentBalance);
        if(currentBalance > savingsTarget)
```

Debugging InterestCalculator2 (3)

Once the breakpoints are in place we can then execute the method as we did before. The method will execute until the first breakpoint and halt, displaying the debugging window.



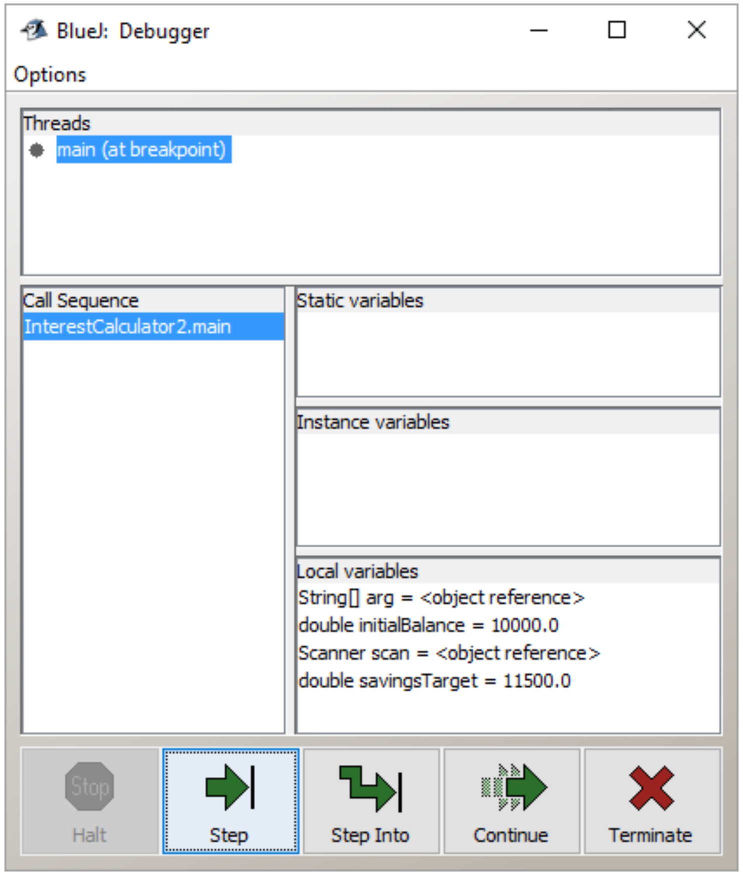
main (at breakpoint)

InterestCalculator2.main

```
String[] arg = <object reference>
double initialBalance = 10000.0
Scanner scan = <object reference>
double savingsTarget = 11500.0
```

Debugging InterestCalculator2 (4)

Continue through the program using the Step / Step Into buttons, taking note of the variables and their values in the bottom right box.



```
String[] arg = <object reference>
double initialBalance = 10000.0
Scanner scan = <object reference>
double savingsTarget = 11500.0
```

Exercise 2 – Roman Numerals

Write a program that converts a positive integer into the Roman number system. The Roman number system has digits **I** (1), **V** (5), **X** (10), **L** (50), **C** (100), **D** (500) and **M** (1000). Numbers up to 3999 are formed according to the following rules:

a) As in the decimal system, the thousands, hundreds, tens and ones are expressed separately.

b) The numbers 1 to 9 are expressed as:

1	**I**	6	**VI**
2	**II**	7	**VII**
3	**III**	8	**VIII**
4	**IV**	9	**IX**
5	**V**		

(An **I** preceding a **V** or **X** is subtracted from the value, and there cannot be more than three **I**s in a row.)

Exercise 2 (2)

c) Tens and hundreds are done the same way, except that the letters **X**, **L**, **C**, and **C**, **D**, **M** are used instead of **I**, **V**, **X**, respectively.

Your program should take an input, such as 1978, and convert it to Roman numerals, MCMLXXVIII.

See JFE, 2nd Ed, exercise P3.26 on p. 130.

Exercise 2 (3): Conversion Algorithm

Hints:

1. Use the following method to convert a positive number into Roman numerals and return it as a string.

   ```
   public static String convert(int  n)
   ```

2. Inside the method, write an **`if`** statement to check for numbers in the range 0 to 3999.

3. If the above is valid, apply the following rules :

   i. Use **`if`** statements to convert numbers in the range 1 and 9 into Roman numerals.

      Note: an **I** preceding a **V** or **X** is subtracted from the value, and there cannot be more than three **I**s in a row.

   ii. Tens and hundreds are done in the same way, using **`if`** statements, except that the letters **X**, **L**, **C** and **C**, **D**, **M** are used.

Exercise 2: Basic structure of Class RomanNumber

```
/* Roman Number system class */
import java.util.Scanner;

public class RomanNumber
{
    public static void main(String[] args)
    {
     /* write code to take an input and store it
        in a variable, number */

     // call method convert
     String romanStr  = convert(number);

     // Output value returned from convert method.
    }
```

Exercise 2: Basic structure of Class RomanNumber (2)

```
public static String convert(int  n)
{
    // convert the number, n, into Roman numerals
    if ((n >= 1) && (n <= 3999))
    {
        /* Step 1 Determine how many ones, tens,
           hundreds and thousands are in the given
           number, n */
        // for n = 1234, ones is 4
        int ones = n % 10;
        // for n = 1234, tens is 3
        int tens = (n / 10) % 10;
        int hundreds = write formula for the hundreds
        int thousands =  write formula for the thousands

     //… see next slide
```

Exercise 2: Basic structure of Class RomanNumber (3)

```
//… continuation of the method convert()
        /* Step 2 Convert the ones into a Roman numeral
                                   and save it in romanOnes*/
        /* Step 3 Convert the tens and save it in romanTens */
        /* Step 4 Convert the hundreds and save it in
                                          romanHundreds */
        /* Step 5 Convert the thousands and save it in
                                        romanThousands */
              return romanThousands + romanHundreds
                                   + romanTens + romanOnes;
           }
           else
           {
              return "Number can’t be converted ";
           }
        }   // end of method convert()
  } // end of class RomanNumber
```

Exercise 2 (Step 2): convert the ones into a Roman numeral

```
String romanOnes;
if (ones == 1)
    romanOnes = "I";
else if (ones == 2)
    romanOnes = "II";
else if (ones == 3)
    romanOnes = "III";
else if (ones == 4)
    romanOnes = "IV";
else if (ones == 5)
    romanOnes = "V";
else if (ones == 6)
    romanOnes = "VI";
else if (ones == 7)
    romanOnes = "VII";
else if (ones == 8)
    romanOnes = "VIII";
else if (ones == 9)
    romanOnes = "IX";
else // (ones == 0)
    romanOnes = "";
```

Exercise 2 (Steps 3-5): convert the tens, hundreds and thousands into Roman numerals

The code required for steps 3, 4 and 5 is similar to that used to convert the ones into a Roman numeral (step 2).

In step 3, you will be replacing the letters **I** (1), **V** (5) and **X** (10) with the letters **X** (10), **L** (50) and **C** (100), respectively.

Step 4 follows the same format, but replacing with the numerals for units: **C** (100), **D** (500) and **M** (1000).

In Step 5, use only **M** (1000) for thousands (from 1 to 3).

Exercise 3

Rewrite the Roman number system program by implementing and using the following method:

```
public static String romanDigit(int digit, String one,
                                String five, String ten)
```

The above method translates one decimal digit, using the three strings specified for the one, five and ten values.

You would call the method as follows:

```
romanOnes = romanDigit(n % 10, "I", "V", "X");
```

Exercise 3 (2)

The Roman number system follows the same pattern for ones, tens and hundreds just as the decimal system does, however, the numerals change. *See slide 19.*

ones	example (7)
I, **V**, **X**	**VII**
tens	example (70)
X, **L**, **C**	**LXX**
hundreds	example (700)
C, **D**, **M**	**DCC**

Your program should use division to separate the ones, tens and hundreds values and send that value, along with the correct numeral pattern to the `romanDigit` method.

Exercise 3 (3)

The `romanDigit` method will receive a single **`int`** value and the corresponding three `String` values.

Example (7):

The **`int`** value `7` is sent to the method, along with the `String` values `"I"`, `"V"` and `"X"`.

The method receives the value 7 in the variable `digit`, the value `"I"` in the variable `one`, the value `"V"` in the variable `five` and the value `"X"` in the variable `ten`. The method should concatenate the variables as follows:

```
    return five + one + one;
```

Exercise 3 (4)

```
int ones = n % 10;
String romanOnes = romanDigit(ones, "I", "V", "X");
```

For instance, when using the value 2018 as `n`, the `romanDigit()` method will receive the value 8 as the first input parameter. The `String` value returned from the method, `"VIII"` (the Roman numeral representing eight) would be stored in the variable `romanOnes`.

Your program should continue as follows:

```
int tens = (n / 10) % 10;
String romanTens = romanDigit(tens, "X", "L", "C");
```

This time, the `romanDigit()` method will receive the value 1 as the first input parameter, representing 1 in 2018.
Note: the other arguments sent to the method have been changed accordingly.

Complete the program by sending the values representing hundreds and thousands.

Marked Exercise 1

See JFE, 2nd Ed, exercise P3.28 on p. 131.

A year with 366 days is called a leap year. Leap years are necessary to keep the calendar synchronised with the sun because the earth revolves around the sun once every 365.25 days. Actually, that figure is not entirely precise, and for all dates after 1582 the *Gregorian correction* applies.

1. Usually years that are divisible by 4 are leap years: for example, 2020.

2. However, years that are divisible by 100 (for example, 1900) are not leap years, but years that are divisible by 400 are leap years (for example, 2000).

Write a program that asks the user for a year and computes whether that year is a leap year. Your program should follow the pattern given in the next slide and available from
http://www.dcs.bbk.ac.uk/~roman/sp1/java/LeapYear.java

Marked Exercise 1 (2)

```
import java.util.Scanner;
public class LeapYear
{
    public static boolean isLeapYear(int year)
    {
        // INSERT YOUR CODE HERE
    }

    public static void printLeapYear(int year)
    {
        System.out.println("Year " + year + " is" +
                    (isLeapYear(year) ? "" : " not") + " a leap year");
    }

    public static void main(String[] args)
    {
        printLeapYear(2020);
        printLeapYear(2000);
        printLeapYear(1900);
        System.out.println("Enter a year: ");
        Scanner scanner = new Scanner(System.in);
        int year = scanner.nextInt();
        printLeapYear(year);
    } // end of method main
} // end of class
```

Home Work
Java for Everyone by C. Horstmann

Read Chapters 2 & 3, which are available online from

http://vufind.lib.bbk.ac.uk/vufind/Record/566484

and complete the following exercises:

- Exercise R2.5
- Exercise R2.6
- Exercise R2.7
- Exercise P2.4
- Exercise R3.4
- Exercise P3.15
- (extra) Exercise P3.14