Java: Code by example

OOP 101

Object Orientated Program right from the beginning.

Version Eight

Author: Steve Eilertsen

Steve.eilertsen@gmail.com

JAVA BACKGROUD

1. For a program coded in Java the **"Java Runtime Environment" (JRE)** needs to be installed on the computer. This creates the Java Virtual Machine (JVM) see below. The JVM translates, line-by-line the Java **bytecode** to machine code. This is why Java is an **interpreted language** (not a compiled language).
2. The JRE creates a **"Java Virtual Machine" (JVM).** This is a virtual machine (computer) created inside the RAM. The JVM compiles the bytecode and then runs the compiled machine code within the JVM. This makes Java a secure language.
3. To code a Java program, one needs the **"Java Development Kit" (JDK).** This compiles the source code (English) to **bytecode** which is saved into a file. This file of bytecode is run by the JVM. Important editions of Java are called "Long Term Support" (LTS) – these editions are JDK8, JDK11, JDK17 and JKD21. We are going to use JDK11.
4. One can code in Notepad, but this is inconvenient and time consuming. Therefore, developers use an **"Integrated Runtime Environment" (IDE)** to assist and speed up development time. The IDE provides useful menus, compiler options, troubleshooting tools, debugging tools, layout tools, auto complete, colour coding etc

JAVA – INTRODUCTORY NOTES:

- In the coding examples that follow the line numbers are for reference only – they are not part of the code.
- Java is **case sensitive** ("public" is not equal to "Public")
- Classes must always start with a capital letter (upper case)
- Methods and variables (identifiers) must start with a small letter (lower case)
- Easy-to-read code is vitally important when we code. We use descriptive class names, descriptive method names and descriptive variable names; layout is equally important i.e. **indentation, and whitespace.**
- The name of the class must match the name of the **file**.
- We code with an OOP two-class model. Only the **UI class** has a main method.

TWO CLASS PROGRAMS using OOP

- The **user interface class (the UI class)** and the **manager class**.
- The user interface (UI class) has the main method.
- The manager class. This has the methods that can be called by the UI class.
- Each class is named using a capital letter.

The User Interface class – Note the indentation and use of open lines (whitespace). This makes your code more readable for someone else.

The UI class has the **main method** and is therefore the **point of entry** into this **multi-class program**.

Lines 1 to 3 are **comments**, although comments can (and should) appear anywhere in a program that they are needed.

The **indentation** makes it clear that there is a **block of code** inside a block of code (block 9 to 15 is inside block 5 to 17).

In line 12 we create an **instance** of "HelloWorld Manager" which is called "myManager".

In line 13 we call the "printThis" **method,** which is found in "myManager".

```
1  // steve eilertsen
2  // hello world - two class
3  // the UI class
4
5  public class HelloWorldUI
6
7  {
8
9     public static void main(String[]args)
10    {
11
12       HelloWorldManager myManager = new HelloWorldManager();
13       myManager.printThis();
14
15    }
16
17 }
```

The Manager class

- Here we find the **methods**. (The class below does not have a constructor method yet.)
- Line 9 below has a void method called "printThis". The output goes to the monitor using **text** mode.
- A method can be **called** any number of times (see line 13 above) – or not called at all.
- From a **template class** like this we can create many **instances** of the class, each with its own name (see line 12 above – "myManager".)

```
 5 public class HelloWorldManager
 6
 7 {
 8
 9    public void printThis()
10    {
11
12       System.out.println("Hello World");
13
14    }
15 }
```

RULES FOR NAMING IDENTIFIERS OR VARIABLES

When we **name** a class, a method or an **identifier** (also called **variables**) . . .

- We may not start with a number.
- Do not use special characters in your naming conventions eg @ # $ % & ! etc
- The class or variable name must not have a space.
- We may not use the Java reserved words when we name our classes or variables.

JAVA RESERVED WORDS

These are words that are "reserved" by the Java language. You cannot use these words as **identifiers (e.g. variable names).**

Abstract, default, goto, package, thi,s assert, do, if, private, throw, Boolean, double, implements, protected, throws, break, else, import, public, transient, byte, enum, instanceof, return, true, case, extends, int, short, try, catch, false, interface, static, void, char, fina,l long, strictfp, volatile, class, finally, native, super, while, const ,float, new, switch, continue, for, null, synchronized

Exercise 1.1. The UI class, the manager class, methods and calling methods.

Using the example above complete the following exercise. Compile and run your program as needed.

1. In jGRASP create a folder called “HelloSchool”.
2. In HelloSchool create a UI class and a manager class – “HelloSchoolUI” and “HelloSchoolManager”.
3. The UI class must have the main method.
4. In the UI class create an instance of HelloSchoolManager.
5. In the manager create two methods i.e. “printThis” and “printThisTwice”.
6. The method printThis must print “Hello School” to the monitor.
7. The method printThisTwice must print “Hello School Again” to the monitor.
8. The **output to the monitor** must look like this . . .

```
Hello School

Hello School Again

Hello School Again
```

INPUT FROM THE KEYBOARD

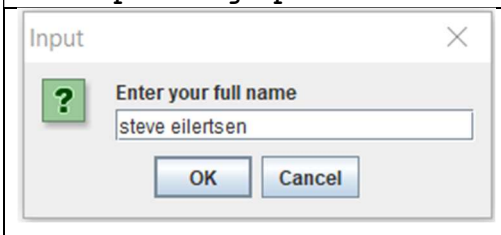
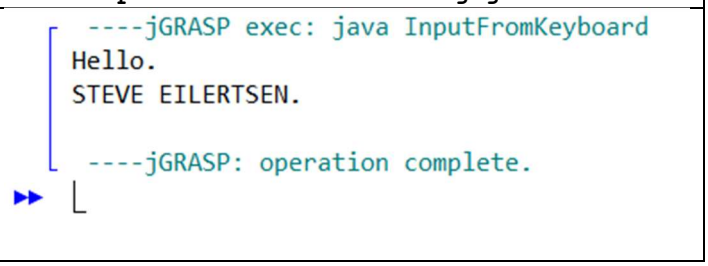
The UI class: We try not to "hardcode" values that can change. Getting input from the keyboard makes a program useful. We need a method to get the **input** (getName) and a method to print the **output** (printName). These methods are **called** in the UI class, but the code is to be found in the manager class.

```
 1 // steve eilertsen.
 2 // Input from keyboard
 3
 4 public class InputFromKeyboard
 5 {
 6
 7       public static void main(String[]args)
 8       {
 9
10          InputFromKeyboardManager nc = new InputFromKeyboardManager();
11          nc.getName();
12          nc.printName();
13       }
14 }
```

==========

The manager class: We are going to import a library to get input from the keyboard using **graphics** mode, and then to display the result using **text** mode – both modes are useful. This library is "**JOptionPane**". We are going to use the method i.e. "**showInputDialog**" We are going to take the input from the keyboard and put it into upper case using the String method "toUpperCase".

```
 4 import javax.swing.JOptionPane;
 5
 6 public class InputFromKeyboardManager
 7 {
 8
 9       // global variable. All methods in the manager can use it.
10       String name = null;
11
12       public void getName()
13       {
14          name = JOptionPane.showInputDialog(null, "Enter your full name");
15          name = name.toUpperCase();
16
17       }
18
19       public void printName()
20       {
21          System.out.println("Hello." + "\n" + name + ".");
22       }
23 }
```

The input in graphics mode using jGRASP	The output in text mode using jGRASP
	

NOTES

In line 10 in the manager class, we are declaring a global variable to store the person's name that comes in from the keyboard. The variable name we are using is "name" – a good descriptive name. **The datatype is "String".** String has a capital letter because string is a class.

We join text together using "+"; see line 12. This is called to "**concatenate**".

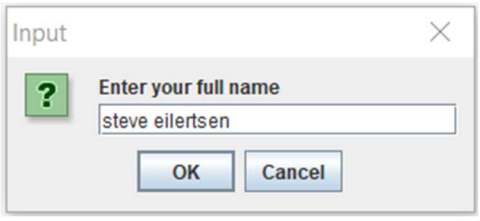
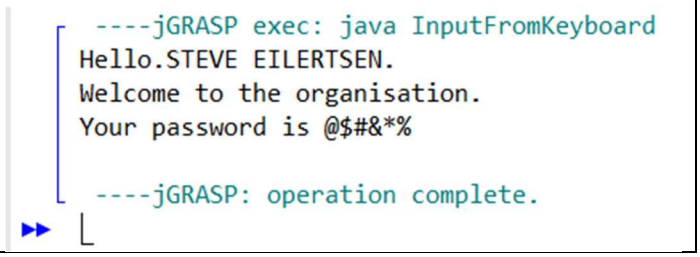
We can format our output by concatenating an escape character to our regular text.

"\n" – The new line character. Moves the text onto a new line; see line 21.

Exercise 1.2. Input from the keyboard.

Using the example above complete the following exercise. Compile and run your program as needed.

1. In jGRASP create a folder called "GetInput".
2. In GetInput create a UI class and a manager class – "GetInputUI" and "GetInputManager".
3. The UI class must have the main method.
4. In the UI class create an instance of GetInputManager.
5. In the manager create four methods i.e. "getName", "formatUpper", "formatLower" and "printThis"
    a. A method to get the name from the keyboard.
    b. A method to change the formatting of the name to uppercase.
    c. A method to change the formatting of the name to lowercase.
    d. A method to print the output to the monitor.
6. The output to the monitor must look like the screen shot below.
7. Copy both your classes to a single notepad file.
8. Print two copies of your notepad file – one to hand in, and one for your own study notes.

The input in graphics mode.	The output in text mode.
	

Output to the monitor in both text mode and graphics mode.

Text mode – the code

```
19        public void printName()
20        {
21           System.out.println("Hello." + name + "." + "\n" + "Welcome" + "\n" +
22           "Your password is @$#&*%");
23        }
```

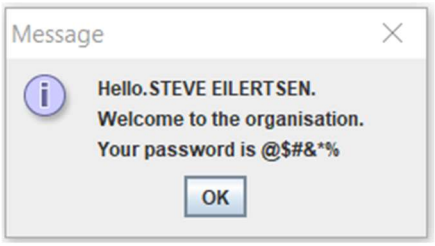
The output

```
 ----jGRASP exec: java InputFromKeyboard
Hello.STEVE EILERTSEN.
Welcome to the organisation.
Your password is @$#&*%

 ----jGRASP: operation complete.
```

Graphics mode – the code

```
25        public void printName2()
26        {
27
28           JOptionPane.showMessageDialog(null, "Hello." + name + "." + "\n" +
29           "Welcome to the organisation." + "\n" +
30           "Your password is @$#&*%");
31        }
```

The output



Exercise 1.3. Program flow.

An **event driven program** is one that allows a user to interact with the program in such a way that the program flow is altered according to the needs of the user. For this reason, program flow, or the "order of things", is critical.

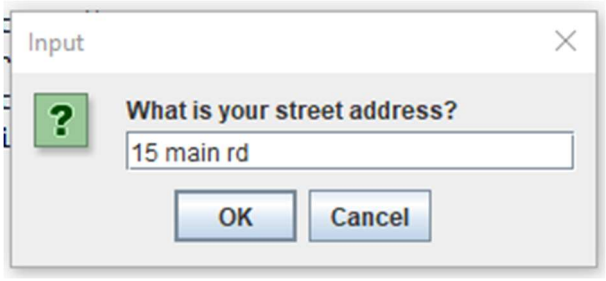
Create your own 2 class program that follows the methods above and follows the dialog below.

Copy and paste your work into a single notepad file, print it and hand it in for marking. If successful you may print a copy for yourself for your study notes.

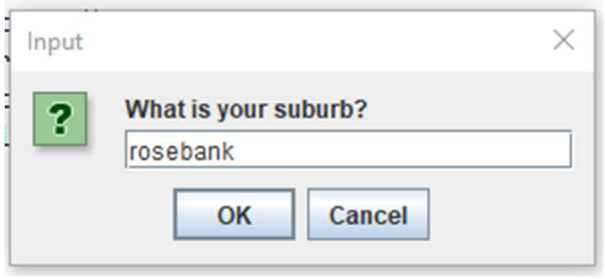
Program Flow: Your program must follow this dialog using graphics mode for both input and output.

1)

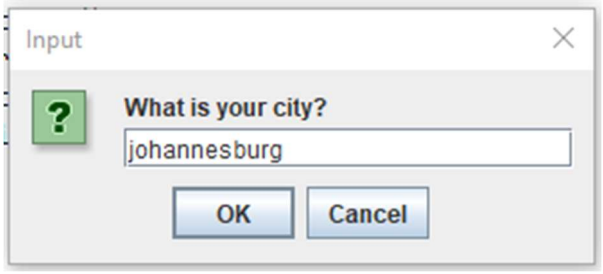
2)



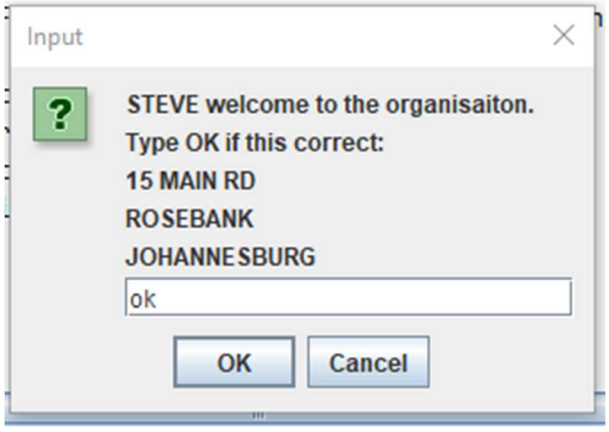
3)



4)



5)



NOTE:
All input from the keyboard has been placed into uppercase. For this use only one method – one method must uppercase name, address, suburb, and city.

If you were to use four separate methods, you will have created **redundant code**. Redundant code is unneeded, unnecessary code that can be replaced using a more concise coding design. Redundant code is also difficult to update, edit and troubleshoot because it appears in more than one place.

Object Orientation Programming (OOP)

OOP is about creating objects. Objects have **properties** (characteristics, features) and **methods** (that do something).

Here is an example of creating an object that we have used before.

```
ManagerInputFromKeyboard myManager = new ManagerInputFromKeyboard();
```

Example: A "person" object. A person can be short or tall (**properties**). A person can play tennis or hockey (**methods**)

The most used object is a **String** object i.e. text in Java is always an object. We can create a String object like this.

```
String name = new String("steve");
```

But we seldom use this syntax. For everyday use we declare String like this . . .

```
String name = "steve";
```

Here we have created a String object called "name". Its properties are steve**.**

The created object "name" has access to many String methods. Here are two examples – the "toUpperCase" method and the "toLowerCase" method.

```
name = name.toUpperCase(); // steve would become STEVE
name = name.toLowerCase(); // STEVE would become steve
```

METHODS.

Methods can be coded for us by the Java language, or we can create our own. Here is an example of a method we coded earlier.

```
12        public void getName()
13        {
14           name = JOptionPane.showInputDialog(null, "Enter your full name");
15           name = name.toUpperCase();
16
17        }
```

We name our created objects or coded methods with a small letter i.e. name, not Name. printThis, not PrintThis.

We recognise a method because the name is followed by **round brackets** e.g. `name.toLowerCase();`

METHOD AND CLASS HEADERS

Do not confuse **method headers** with **class headers.**

Method header – made up of 4 parts	Class header – made up of 3 parts
`public void getName()`	`public class ManagerInputFromKeyboard`

In the table below the differences between the **class header** and the **method header.**

	A - The access modifier.	B – The return type.	C	D – the name of the class or method.	E – The arguments.
Method header	public	void	N/A	myMethod	( )
Class header	public	N/A	class	MyClass	N/A

In line one is the method header. In line two is the class header.

Column A – The **access modifier** can be public, private, or protected.

Column B - Only a method has a **return type** – void or String or int or double or an array etc.

Column D – Classes are named with a capital letter and methods (and variables) named with a small letter.

Column E - Only a method has **arguments** – also often generally called **parameters**. In this example the argument list is empty.

THE CONSTRUCTOR METHOD

A class from which you **instantiate objects** should have a special method called the **constructor.** Its primary task is to **initialize the properties** of the new created object. For this reason, the constructor runs **automatically** when you create the object.

Constructor methods have two characteristics:

1) The have the same name as their class.
2) They do not have a return type like other methods (not void, not String . . . No return type)

Consider the manager class below that has a constructor method.

```
 1 // steve eilertsen
 2 // Input from keyboard manager
 3
 4 import javax.swing.JOptionPane;
 5
 6 public class InputFromKeyboardManager
 7 {
 8
 9       // global variables. All methods can use them.
10       String name = null;
11       String email = null;
12       String memberSt = null;
13       boolean member = false;
14
15       public InputFromKeyboardManager()
16       {
17
18          name = JOptionPane.showInputDialog(null, "Enter your full name");
19          name = name.toUpperCase();
20
21          email = JOptionPane.showInputDialog(null, "Enter your email address");
22
23          memberSt = JOptionPane.showInputDialog(null, "Member? Type true/false.");
24          member = Boolean.parseBoolean(memberSt);
25
26       }
27
28       public void printName()
29       {
30          System.out.println("Hello." + name + "." + "\n" + "Welcome." + "\n" +
31          "Your email address is: " + email + "\n" + "You are a member: " + member);
32       }
33
34 }
```

NOTES:

1) The constructor method can be found in lines 15 to 26. This method runs automatically when the object is instantiated by a line of code in the UI class.
2) All input from the keyboard arrives as String. Therefore “memberSt” must be parsed to a boolean in line 24.
3) The method “printName” is a normal method and therefore has a return type. In this case “void”.

Grade 10 Java coding task. Floor paint version 1.

Use OOP (a UI class and a Manager class) to solve the following problem.

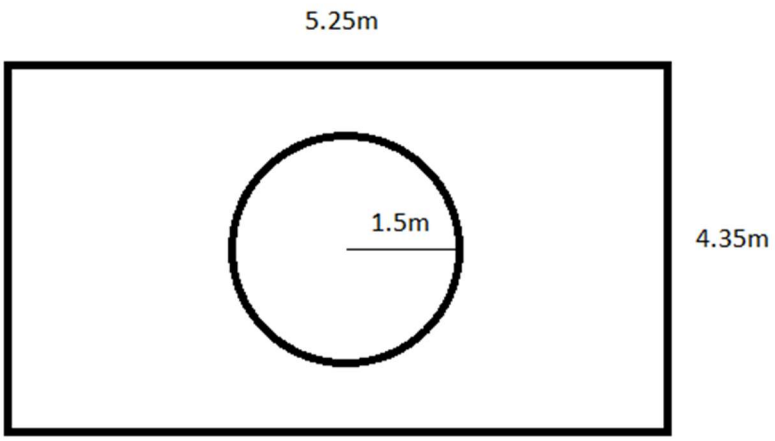
"What is the price of the floor paint needed to paint a specific floor plan?"

NOTE: For this task your manager class may print prompts and answers to the monitor (we should, however, use the UI class for this purpose).

You need to paint a floor with a special floor paint. The middle of the floor has a special wooded circular inlay and does not need any paint – only the floor surrounding the circular shape. Using the dimensions in the diagram below determines the following.

1. The square meterage of the floor to be painted.
2. The number of tins of paint needed
3. The cost of the paint.

Your program must also work for other similar floor shapes and therefore all measurements must come in from the keyboard.



1 square meter requires 350ml of floor paint.

Paint comes in 5-liter tins and 2-liter tins. How many tins should you purchase?

The cost of the 5-liter tin is R574-00 and the 2-liter tin is R256-00 each.

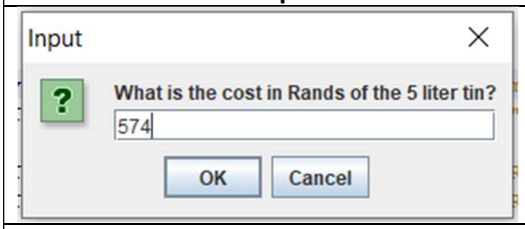
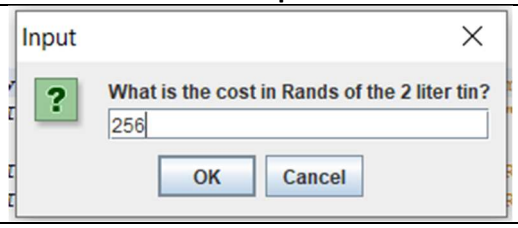
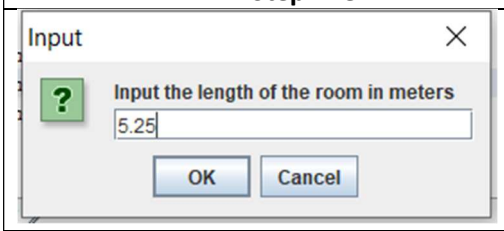
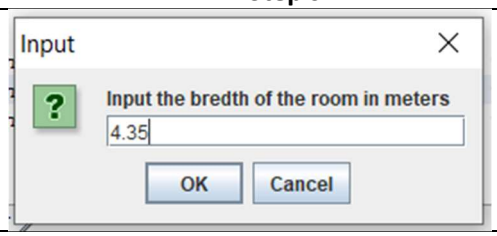
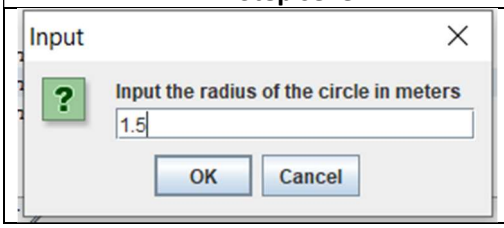
How much will the paint cost?

Phase one. Input. Getting the input from the keyboard

Your program should look like this. Use the **IPO model** i.e. input, processing and output.

The **flow** of your program should logically look like this.

Use the **constructor method** to capture the input from the keyboard.

Step one	Step two
	
Step three	**Step four**
	
Step five	**Step six**
	
Step seven	
	

You need to declare all the **variables** you need for the program. Declare them in the manager class, at the top, above the methods, so that **all** the methods can use them. We call these **global variables**.

```
// Global variables are as follows.

// Some of the String variables we need.

String firstName, lastName;

String lengthSt = null, breadthSt = null, radiusSt = null;

String liter5St = null, liter2St = null;


// Some of the number variable we will need. When we convert the Strings variables to
numbers we need number variables.

double length = 0.0, breadth = 0.0, radius = 0.0;

double liter5 = 0.0, liter2 = 0.0;
```

Problem: With input from the keyboard, Java works with Strings (text). See example below.

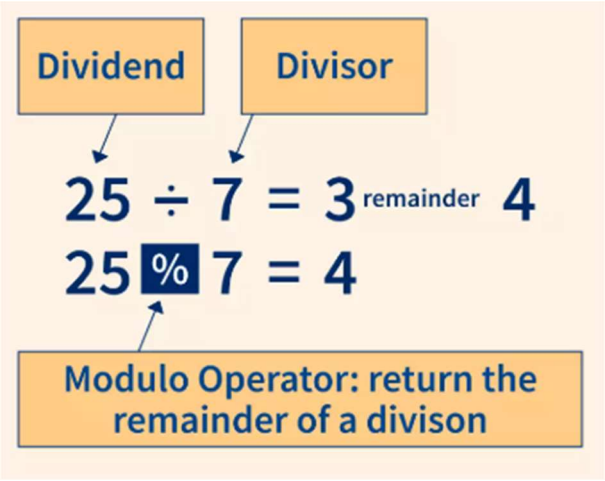
```
firstName = JOptionPane.showInputDialog(null, "Enter your first name");
```

To use numbers (both integers and real) we must convert the String text to a number.

To convert String to integers we use the following . . . Integer.parseInt(String goes here)	
`length = Integer.parseInt(lengthSt);`	NB: This will not work here because we need our variables to be doubles (real numbers), not integers.
To convert String to real numbers we use the following . . . Double.parseDouble(String goes here)	
`length = Double.parseDouble(lengthSt);`	NB: This will work here because we need our variables to be doubles (real numbers).

Working with decimal fractions – how much is left over? What is the remainder?

How to determine the value of the remainder when doing division. Java uses "%" to indicate the modulo operator. SQL uses "MOD".



Phase two. Processing. Determine and check the answers to each calculation.

Now we need to create relevant methods in the manager class to process the input from the keyboard; this will determine the answer.

***Using arithmetic*, the calculation will look like this . . .** We must now code this in Java.

- For this task all our methods will have a **return type** of "**void**" as they will print their own output to the monitor (which is not good, but ok for now)
- For this task all our variables will be **global** so that all methods can have access to them.
- We will need 7 methods to calculate the cost; each method does **one specific task**.

1) Area of the rectangle = L x B. In Java this will be method one.

5.25 x 4.35

Ans = 22.8375m^2

Use these answers to check the accuracy and logic of your coding.

Yes, there will be tiny differences because I rounded off the answers from my handheld calculator.

2) Area of the circle = πr^2 . In Java this must be method two.

3.1416 x (1.5 x 1.5)

3.1416 x 2.25

Ans = 7.0686 m^2

3) Subtract the area of the circle from the area of the rectangle. Method three.

22.8375 – 7.0686

Ans = 15.7689

4) Determine how much paint is needed per square meter. Another method.

15.7689 x 0.35 // here we convert 350ml to whole liters.

Ans = 5.519115 liters

5) Determine how many 5-liter tins will be needed. Another method.

5.519115 ÷ 5 // how many whole 5-liter tins are needed?

Ans = One 5-liter tin

6) Is there a remainder? Another method in Java.

If there is, add a two-liter tin to cover that. To determine the remainder, we use the modulo operator "%"

5.519115 % 5 // this determines the remainder only.

Ans = 0.519115 // Because there is a remainder, not all of the floor has been painted.

If remainder is > 0, add one 2-liter tin of paint to cover the last of the floor

Final answer is one 5-liter tin and one 2-liter tin.

7) Determine the cost. Method seven in Java.

Cost will be 1 x R574.00 plus 1 x R256.00 = R830.00

Seven methods in the manager class should look like this . . .

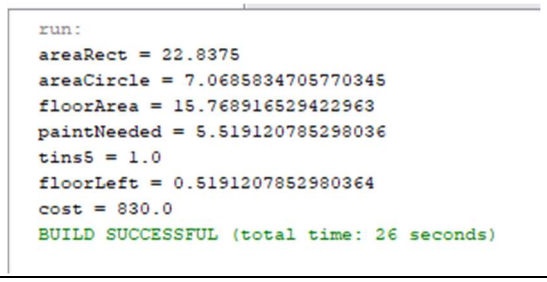
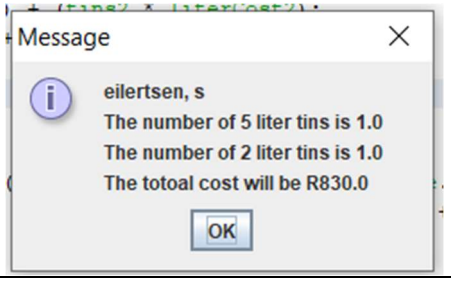
```
public void areaRectangle()
{
    areaRect = length * breadth;    // here a new variable "areaRect" was needed.
    System.out.println("areaRect = " + areaRect);
}
//2
public void areaCircle()
{
      Your code goes here
}
//3
public void subtractArea()
{
      Code goes here
}
//4
public void paint()
{
       Code goes here
}
//5
public void liter5()
{
      Code goes here
}
//6
public void remainder()
{
       Code goes here
}
//7
public void cost()
{
      Code goes here
}
```

Reminders:
If you don't call a method, it does not run. Also be sure to call the methods in the correct order.

You will have to declare several **new variables** to store the calculated values.

If you place a “System.out.println” into every one of your methods, you will get the same output as that shown below. From this output you can see that a lot of **new variables** had to be declared to store the calculated values.

To avoid **logic errors**, we must know all the answers beforehand; that way we know that our code is logically correct. Then we can use our program with other variables and know that the answer will be reliable.

The output shown below is in **text mode**.	Here is the same output shown in **graphics mode**.
	

```
run:
areaRect = 22.8375
areaCircle = 7.0685834705770345
floorArea = 15.768916529422963
paintNeeded = 5.519120785298036
tins5 = 1.0
floorLeft = 0.5191207852980364
cost = 830.0
BUILD SUCCESSFUL (total time: 26 seconds)
```

The solution - We use the bare bones approach

A working version one, then a working version two, then a working version three

THE FLOOR PAINT TASK. VERSION ONE. The UI class and the manger class

```
/*
 * Steve Eilertsen
 */

package floorpaint;

public class FloorPaintUI {

    public static void main(String[] args) {

        FloorPaintManager fm = new FloorPaintManager();

        fm.areaRectangle();
        fm.areaCircle();
        fm.subtractArea();
        fm.paint();
        fm.liter5();
        fm.remainder();
        fm.cost();
    }

} // end of the UI class
```

==========

```
/*
 * Steve Eilertsen
 */
package floorpaint;
import javax.swing.JOptionPane;

public class FloorPaintManager {

    // Global variables are as follows.
    // The Strings we need.
    String firstName, lastName;
    String lengthSt = null, breadthSt = null, radiusSt = null;
    String liter5St = null, liter2St = null;

    // When we convert the Strings to numbers.
    double length = 0.0, breadth = 0.0, radius = 0.0;
    double liter5 = 0.0, liter2 = 0.0;

    // the constructor method. All measurements in meters
       public FloorPaintManager()
    {

        firstName = JOptionPane.showInputDialog(null, "Enter your first name");
        lastName = JOptionPane.showInputDialog(null, "Enter your last name");

        liter5St = JOptionPane.showInputDialog(null, "Cost of the 5 liter tin?");
        liter2St = JOptionPane.showInputDialog(null, "Cost of the 2 liter tin?");

        lengthSt = JOptionPane.showInputDialog(null, "Input the length of the room");
        breadthSt = JOptionPane.showInputDialog(null, "Input the bredth of the room");
        radiusSt = JOptionPane.showInputDialog(null, "Input the radius of the circle");

      // parse the Strings to numbers (doubles) so that we can calculate.
        length = Double.parseDouble(lengthSt);
        breadth= Double.parseDouble(breadthSt);
        radius = Double.parseDouble(radiusSt);
    } // end of the constructor method
```

```
   //The methods we will need later . . .
    public void areaRectangle()
    {

    }
    //2
    public void areaCircle()
    {

    }
    //3
    public void subtractArea()
    {

    }
    //4
    public void paint()
    {

    }
    //5
    public void liter5()
    {

    }
    //6
    public void remainder()
    {

    }
    //7
    public void cost()
    {
    }

}
```

==========

THE FLOOR PAINT TASK

VERSION TWO

```
/*
 * Steve Eilertsen
 */
package floorpaint;


public class FloorPaintUI {


    public static void main(String[] args) {


        FloorPaintManager fm = new FloorPaintManager();


        fm.areaRectangle();
        fm.areaCircle();
        fm.subtractArea();
        fm.paint();
        fm.liter5();
        fm.remainder();
        fm.cost();
    }


} // end of UI class
```

==========

```
/*
 * Steve Eilertsen
 */
package floorpaint;
import javax.swing.JOptionPane;

public class FloorPaintManager {
    //global variables
    String firstName, lastName;
    String lengthSt = null, breadthSt = null, radiusSt = null;
    double length = 0.0, breadth = 0.0, radius = 0.0;
    double areaRect = 0.0, areaCircle = 0.0;
    double floorArea = 0.0;
    double paintNeeded;
    double tins5;
    double tins2;
    double floorLeft;
    double theCost;
    String literCost5St;
    String literCost2St;
    double literCost5;
    double literCost2;
```

```
// the constructor method
public FloorPaintManager()
    {
        firstName = JOptionPane.showInputDialog(null, "Enter your first name");
        lastName = JOptionPane.showInputDialog(null, "Enter your last name");

        liter5St = JOptionPane.showInputDialog(null, "Cost of the 5 liter tin?");
        liter2St = JOptionPane.showInputDialog(null, "Cost of the 2 liter tin?");

        lengthSt = JOptionPane.showInputDialog(null, "Input the length of the room");
        breadthSt = JOptionPane.showInputDialog(null, "Input the bredth of the room");
        radiusSt = JOptionPane.showInputDialog(null, "Input the radius of the circle");

        length = Double.parseDouble(lengthSt);
        breadth= Double.parseDouble(breadthSt);
        radius = Double.parseDouble(radiusSt);
        literCost5 = Double.parseDouble(literCost5St);
        literCost2 = Double.parseDouble(literCost2St);
    } // end of the constructor method

    public void areaRectangle()
    {
        areaRect = length * breadth;
        System.out.println("areaRect = " + areaRect);
    }

    //2
    public void areaCircle()
    {
        areaCircle = Math.PI * Math.pow(radius, 2);
        System.out.println("areaCircle = " + areaCircle);
    }
    //3
    public void subtractArea()
    {
        floorArea = areaRect - areaCircle;
        System.out.println("floorArea = " + floorArea);
    }
```

```
    //4
    public void paint()
    {
        paintNeeded = floorArea * 0.35;
        System.out.println("paintNeeded = " + paintNeeded);
    }

    //5
    public void liter5()
    {
        tins5 = paintNeeded / 5;
        tins5 = Math.round(tins5);
        System.out.println("tins5 = " + tins5);
    }

    //6
    public void remainder()
    {
        floorLeft = paintNeeded % 5;
        System.out.println("floorLeft = " + floorLeft);
        if(floorLeft > 0)
            tins2 = 1.0;
    }

    //7
    public void cost()
    {
        theCost = (tins5 * literCost5) + (tins2 * literCost2);
        System.out.println("cost = " + theCost);
    }

}
```

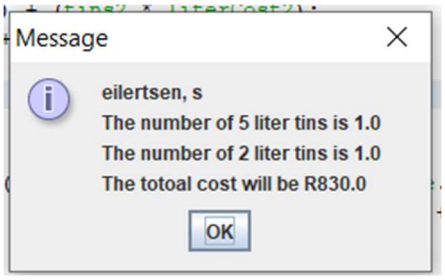
==========

THE FLOOR PAINT TASK

VERSION THREE – This adds the report method to the manager class

We can use "JOptionPane.showMessageDialog" to create a graphically based output report.

Here is an example for you to follow. Yes, it is not perfect, but it will do for now.



Can you list four shortcomings of the dialog box alongside?

_____

_____

_____

_____

```
public void report()
 {
      JOptionPane.showMessageDialog(null, lastName + ", " + firstName.charAt(0) + "\n" + "The
      number of 5 liter tins is " + tins5 + "\n" + "The number of 2 liter tins is " + tins2 +
      "\n" + "The total cost will be R" + theCost);
 }
```

NOTES and reminders for exams.

- We use the method "**showMessageDialog**" to create a mini report in graphics mode.
- We join Strings together using the concatenation operator "+" .
- We also use the concatenation operator "+" to create new sentences made up of Strings and variables joined together.
- We get a new line by using the escape character "\n".
- The code snippet "`firstName.charAt(0)`" gives you the first character (only) of the first name. This is because computers count from zero. The method **charAt(0)** is a String handling method that returns the single character at the specified position – in this case position zero.
- The coder must type the rand sign "R" as part of the output message.
- Rounding to 0,1 or 2 decimal places should be part of this program but we will not tackle this problem right now.

STRING AND STRING METHODS.

We have already seen toUpperCase and toLowerCase. These are two of many String methods used to manipulate Strings. We will use the String "steve eilertsen" as an example. In the box below we will use String methods to separate the first name from the last name. In addition the output of each will be in upper case. The String methods we will use will be toUpperCase, indexOf and two versions of substring

<table>
<tr><th>Input – graphic mode</th><th>Output – text mode</th></tr>
<tr><td> 
</td><td>

```
----jGRASP exec: java StringMethodsUI
Your name is 15 characters long.
Your first name is STEVE
Your surname is EILERTSEN

----jGRASP: operation complete.
```

</td></tr>
<tr><td>We find the position of the space.
Everything before the space is the first name.
Everything after the space is the surname.
The space is not part of the first name or surname.

Strings have index numbers starting from zero
"S" is zero, "t" is one, "e" is two, "v" is three, "e" is four, the space is five.

The String method substring allows us to break Strings into pieces.

name.substring(0, 5) – here substring works with two indexes – the first one is INCLUDED and the second is EXCLUDED.
Therefore substring(0,5) gives us "steve" only.

name.substring(6) – here substring works with one index – the first one is INCLUDED and because it does not have a second index it goes all the way to the end of the name
Therefore substring(6) gives us "eilertsen" only.</td><td>

Indexes									
0	1	2	3	4	5	6	7	8	etc
s	t	e	v	e		e	i	l	etc

The space is index 5.
Therefore, the surname starts at 5 + 1 – one after the space which is index 6.

This is what the codes looks like . . .

```
space = name.indexOf(" ");
// Finds the position of the space.
"space" has been declared as a primitive
datatype - in this case an int (an
integer - a whole number).

firstName = name.substring(0,space);
// Gives us steve only

surname = name.substring(space + 1);
// Gives us eilertsen only
```

</td></tr>
</table>

In the program that follows note the correct use of the comments section. Read it because it outlines what the program does.

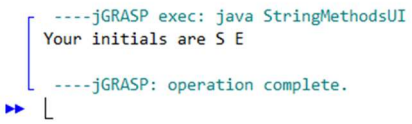
The line numbers below are for teaching purposes only as there are two DIFFERENT classes when we code them.

```
1 // Accepts a full name, determines its length, splits the first from the surname.
2 // Introduction to String methods
3 // The UI class
4
5 public class StringMethodsUI
6 {
7
8    public static void main(String[]args)
9
10   {
11
12      StringMethodsManager myManager = new StringMethodsManager();
13
14      myManager.getNames();
15      myManager.lengthName();
16      myManager.printNames();
17
18   }
19
20 }
22 ==================== New Class ====================================================
23
24 import javax.swing.JOptionPane;
25
26 public class StringMethodsManager
27 {
28    // Global variables to store the values we need
29    private String name, firstName, surname;
30    private int space = 0;
31    private int nameLength = 0;
32
33    public void getNames()
34    {
35       name = JOptionPane.showInputDialog(null, "Enter your name - first and last");
36       name = name.toUpperCase();
37       space = name.indexOf(" "); // find the position of the space
38       firstName = name.substring(0,space); // first name is up to the space
39       surname = name.substring(space + 1); // surname is beyond the space
40
41    }
42
43    public void lengthName() // determines the length of the name in characters
44    {
45       nameLength = name.length();
46
47    }
48
49    // Here below the "+" joins strings together (concatenates)
50    public void printNames()
51    {
52       System.out.println("Your name is " + nameLength + " characters long.");
53       System.out.println("Your first name is " + firstName);
54       System.out.println("Your surname is " + surname);
55
56    }
57
```

Exercise 1.4

Study the program on the previous page. Study it line by line. Understand it 100% before starting the exercise.

Write your own two class program that will determine the initials from a person's name.

Input in graphics mode	Output in text mode	Hints
		Find the space in the name. Find the first name and the surname. Use substring to isolate the first letter of each name. You will need to declare some String variables to store each initial so that you can print them later.

On the pages that follow you will find a list of commonly used String methods.

STRING – The data type and 19 String methods we use to manipulate them.

Java prefers to work with text. The data type for text is **String**.

String is a class – it is not a primitive data type like int (integer) or double (real numbers).

String is a complex data type i.e. a String has methods. We use methods to manipulate Strings.

Method	Description	Input - Arguement	Output - Return type

1) charAt()	Returns the character at the specified index (position)
	
	`char c = myStr.charAt(0)`
	
	`The datatype (char),the variable name(c), object, the method, the arguement`
		int	char
			
			One character only

2) compareTo()	Compares two strings lexicographically (dictionary order)
	
	`int x = myStr1.compareTo(myStr2)`
	
	`The datatype (int),the variable name(x), object, the method, the arguement`
		Strings	int
			
			Whole numbers only

3) compareToIgnoreCase()	Compares two strings lexicographically, ignoring case differences `int x = myS1.compareToIgnoreCase(myS2)`	String	int

4) concat()	Appends a string to the end of another string
	
	`String s = firstName.concat(lastName)`
	
	`Datatype(String),the variable name(s), object, the method, the arguement`
		String	String

5) contains()	Checks whether a string contains a sequence of characters
	
	`boolean b = myStr.contains("Hi")`
	
	`Datatype (boolean),the variable name(b), object, the method, the arguement`
		String	boolean
			
			true or false

6) endsWith()	Checks whether a string ends with the specified character(s) `boolean b = myStr.endsWith("tion")`	String	boolean
7) equals()	Compares two strings. Returns true if the strings are equal, and false if not `boolean b = myStr1.equals(myStr2)`	String	boolean

Method	Description	Input - Arguement	Output - Return type
8) equalsIgnoreCase()	Compares two strings, ignoring case considerations `boolean b myS1.equalsIgnoreCase(myS2)`	String	boolean

9) indexOf()	Returns the position of the first found occurrence of specified characters in a string
	`int x1 = myStr.indexOf("p")`
	`int x2 = myStr.indexOf("planet")`
		String	int

10) lastIndexOf()	Returns the position of the last found occurrence of specified characters in a string `int x =myStr.lastIndexOf("planet")`	String	int
11) length()	Returns the length of a specified string `int x = txt.length()`	String	int
12) replace() replaceFirst() replaceAll()	Searches a string for a specified value, and returns a new string where the specified values are replaced `String s = myStr.replace('l', 'p')`	Two characters	String
13) split()	Splits a string into an array of substrings `String[] arrOfStr = str.split("@", 2)`	Delimiter followed by the limit	String[ ]
14) startsWith()	Checks whether a string starts with specified characters. `boolean b = myStr.startsWith("o")`	String	boolean

15) substring()	Returns a new string which is the substring of a specified string
	`String s1 = myStr.substring(4)`
	`// index 4 to the end`
	`String s2 = myStr.substring(4,7)`
	`// index 4 to 7 only – include, exclude`
		One or two index values	String

16) toLowerCase()	Converts a string to lower case letters `String s = txt.toLowerCase()`	Nil	String
17) toString()	Returns the value of a String object `String s = txt.toString()`	Nil	String

Method	Description	Input - Arguement	Output - Return type
18) toUpperCase()	Converts a string to upper case letters `String s = txt.toUpperCase()`	Nil	String
19) trim()	Removes whitespace from both ends of a string. `String s = myStr.trim()`	Nil	String

PRIMITIVE DATA TYPES

In the exercise 1.4 above we used the **primitive data type** int (integer – a whole number) to find the position of the space. For completeness here is a table of the other commonly used primitive datatypes.

Datatype	Java	Examples – declare and initialize

integer – whole numbers	**int**
	32 bits
		int number = 7;
		int start = 0;
		int number2 = -76;
		We declare it and we initialize it.
		See lines 30 and 31 on page 9.

double – real numbers	**double**
	64 bits
		double number = 7.5;
		double zero = 0.0;
		double number2 = -5.6567;
		We declare it and we initialize it.

boolean – true or false	**boolean**
	1 bit
		boolean value1 = true;
		boolean value2 = false;
		We declare it and we initialize it.

character – one single character	**char** **16 bits**	char letter = 'A'; char letter2 ='a'; char myCharacter = '&'; NOTE: Single inverted commas for char. We declare it and we initialize it.

CLASS HEADERS AND METHOD HEADERS

In the table below the differences between the **class header** and the **method header.**

	A - The access modifier.	B – The return type.	C	D – the name of the class or method.	E – The arguments.
1)	public	N/A	class	MyClass	N/A
2)	public	void	N/A	myMethod	( )

In line one above is the class header. In line two is the method header.

Column A – The access modifier can be **public**, **private**, or **protected**.

Column B - Only a method has a **return type** – void or String or int or double etc.

Column D – Classes are named with a capital letter and methods (and variables) named with a small letter.

Column E - Only a method has **arguments** – also often generally called **parameters**. In this example the argument list is empty.

DATE AND TIME – creation of a date time object using now( )

Let's create another type of object – a **date** object and a **time** object.

Here is a single method (at this stage) on how to create a date object and a time object.

Java has an import library for dates and a separate library for time.

```
import java.time.LocalDate; // Note: Date is in the "time" folder
import java.time.LocalTime;
```

We do NOT create a date object or a time object with the "new" keyword. The "**now( )**" method give us the date now or the time now. This information comes from the OS.

```
LocalDate theDate = LocalDate.now();
LocalTime theTime = LocalTime.now();
```

Date objects output like this. `2024-03-21` // yyyy-MM-dd

Time objects output like this. `12:31:03.999` // hh:mm:ss:nn (there are 1000 nanoseconds to one second)

When discussing date and time "**MM**" is used for months and "**mm**" is used for minutes.

CONDITIONAL STATEMENTS

Programs are all about making decisions and we use the "**If**" statement for this. An "if" statement with an "**else**" statement chains them together. Multiple "if" statements without an "else" will be independent and standalone.

Commentary	Condition(s) coded in Java

1)
 	
If a condition is true, this happens (line 3)
 	
If the condition is false, nothing happens (line 3 does not execute)
	`1 if (price > 1000)`
	`2 {`
	`3  discount = 50;`
	`4 }`
	
	`Line 3 may execute.`
	`Line 3 may not execute.`

2)
 	
If condition1 is true, this happens.
 	
If condition2 is true, that happens.
 	
Both could execute or neither could execute or only one may execute.
 	
Standalone "if" statements not chained together must be used with great caution and they result can be unexpected (called **logic errors**)
	`1 if (total > 500)`
	`2 {`
	`3  discount = 50;`
	`4 }`
	`5`
	`6 if(total > 1000)`
	`7 {`
	`8  discount = 70;`
	`9 }`
	
	`Line 3 could execute.`
	`Line 8 could execute.`
	`Line 3 and line 8 could BOTH execute.`
	`Neither line 3 nor line 8 may execute.`

3)
 	
If condition1 is true, line 33 executes.
If condition1 is false, then line 38 will execute.
	`31 if (total < 500)`
	`32 {`
	`33  discount = 50;`
	`34 }`
	`35`
	`36 else`
	`37 {`
	`38  discount = 70;`
	`39 }`
	
	`One WILL execute – either 33 or 38`

4)
 	
If condition1 is true, line 33 executes.
 	
If condition1 is false, then condition2 MAY execute.
If condition2 is true, line 38 executes.
 	
If condition1 and condition2 are both false, neither line 33 nor line 38 will execute.
	`31 if (total > 500)`
	`32 {`
	`33  discount = 50;`
	`34 }`
	`35`
	`36 else if(total > 1000)`
	`37 {`
	`38  discount = 70;`
	`39 }`
	
	`Line 33 OR line 38 could execute but never both.`
	`If line 33 executes then line 38 will never execute.`

Commentary	Condition(s) coded in Java

5) If condition1 is true, then line 33 will execute. The conditions in line 36 and line 40 will never be tested. Because they are chained together more than one cannot execute.
 	
total = 5000.
The discount of 100 will never be offered because line 31 will already have executed.
 	
total = 100
None of the lines (33, 38 or 42) will execute.
	31 if (total > 500)
	32 {
	33 discount = 50;
	34 }
	35
	36 else if (total > 1000)
	37 {
	38 discount = 70;
	39 }
	40 else if (total > 2000)
	41 {
	42 discount = 100;
	43 }

6) total = 2000.
 	
All three conditions will execute as they are separate standalone statements.
 	
Discount will be 50, but then discount will be 70, and finally discount will be 100. This is not good and is called “redundant” because lines of coding are running that do not need to run, and the discount should be 100 right from the start.
	31 if (total > 500)
	32 {
	33 discount = 50;
	34 }
	35
	36 if (total > 1000)
	37 {
	38 discount = 70;
	39 }
	40 if (total > 2000)
	41 {
	42 discount = 100;
	43 }

7) Because they are chained together ONLY one line can execute – line 35 or line 40 or line 44 or line 48.
 	
Because they are chained together ONE line MUST execute – line 35 or line 40 or line 44 or line 48.
 	
NOTE: We use “equals” when comparing String to see if they are the same
	33 if(house.equals("Ibus"))
	34 {
	35 colour = "red";
	36 }
	37
	38 else if (house.equals("Cormorant"))
	39 {
	40 colour = "Blue";
	41 }
	42 else if (house.equals("Kingfisher"))
	43 {
	44 colour = "Green";
	45 }
	46 else
	47 {
	48 colour = "Unknown";
	49 }

8) Because they are chained together ONLY one line can execute – line 35 or line 40 or line 44 or line 48.
 	
Because they are chained together ONE line MUST execute – line 35 or line 40 or line 44 or line 48.
 	
NOTE:
We use “==” when comparing numbers to see if they are the same.
We use “=” as the assignment statement – to assign a value to a variable
	33 if(number == 10)
	34 {
	35 prize = "First Prize";
	36 }
	37
	38 else if (number == 20)
	39 {
	40 prize = "Second Prize";
	41 }
	42 else if (number == 30)
	43 {
	44 prize = "Third Prize";
	45 }
	46 else
	47 {
	48 prize = "Consolation Prize";
	49 }

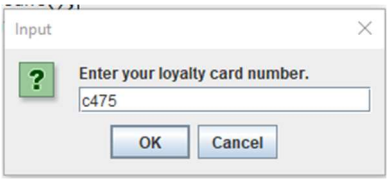
```
// Conditional statement. Also date and time. If login name from the keyboard
// is the same as the login name coded into the program - access granted.

public class LoginConditionalUI
{
   public static void main(String[]args)
   {
      LoginConditionalManager myManager = new LoginConditionalManager();
      myManager.getDateTime();
      myManager.enterName();
      myManager.login();
   }
}
=========================================================================
// Login manager with date and time

import javax.swing.JOptionPane;
import java.time.LocalDate;
import java.time.LocalTime;

public class LoginConditionalManager
{
   // global variables for all methods to use
   private String myName = "steve";
   private String name = null;
   private LocalDate theDate = null;
   private LocalTime theTime = null;

   public void getDateTime()
   {
      theDate = LocalDate.now();
      theTime = LocalTime.now();
      System.out.println("The date is " + theDate);
      System.out.println("The time is " + theTime);
   }

   public void enterName()
   {
            name = JOptionPane.showInputDialog(null, "Enter your name");
            name = name.toLowerCase();
   }

   public void login()
   {
      if(name.equals(myName)) // Note the indentation and layout
      {
         System.out.println("Access granted");
      }
      else
      {
         System.out.println("Access denied");
      }
   }
}
=============================================================
SAMPLE OUTPUT
   | The date is 2024-03-22
   | The time is 08:57:59.054
   | Access granted
```

Exercise 1.5A

Study the program on the previous page. Study it line by line. Understand it 100% before starting the exercise. You will need to use String handling methods and a conditional if statement to complete this exercise.

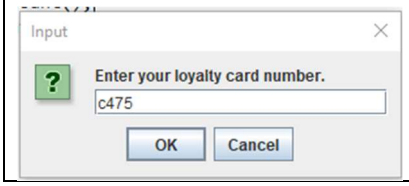
Write your own two class program that will grant a discount based on the persons loyalty card number.

1. Loyalty cards that begin with "S" are granted a 10% discount. E.g. S786
2. Loyalty cards that begin with "C" are granted a 7% discount. E.g. C978
3. Loyalty cards with a number longer than 5 characters get 12% E.g. 788736
4. Every other loyalty card gets a 5% discount. E.g. 7865

Input in graphics mode	Output in text mode	Hints
	Loyalty manager ========== Your loyalty card number is C475 Your discount is 7%	The card number must be a String so that you can use String handling methods. You will need to use conditional "if else" statements to make sure that only ONE discount can apply. You will need to declare some String and integer variables so that you can print them later. Note the percentage sign in the output.

Exercise 1.5B

Edit your program to print out the date and time when the discount was granted.

Input in graphics mode	Output in text mode	Hints
	Loyalty manager ========== Date and time: 2024-03-22 / 07:43:07.413 Your loyalty card number is C475 Your discount is 7%	You will need to declare date and time variables that are global so that all the methods will have access to their values. Compile your manager class before you try to run your program.

CONDITIONAL STATEMENTS – Multiple compound conditions.

We are not limited to one condition – we can join conditions together using the logical operators.
Examples in order of **precedence**

- if it is not assembly day so learners can wear short pants – the **NOT** logical operator. (!)
- if it is Friday and it is summer so learners can wear t-shirts – the **AND** logical operator. (&&)
- if it is Friday or it is raining so learners can wear a dri-mac – the **OR** logical operator. (||)

Java examples – note that the variable must be repeated on each side of the logical operator.

- If(value != 7 && value > 10) { } // value is not equal to 7 and value is larger than 10
- If(value > 10 || value < 0) { } // value is larger than 10 or value is smaller than 0
- If(name.equals("steve") && password.equals("123") // name is steve and password is 123
- If(total == 100) || prizeAwarded == false) // total is equal to 100 or prize has not been awarded

Precedence – like BODMAS, the logical operators are executed in the order of NOT, AND and finally OR

DATE AND TIME – Creating our own date or time.

Beside the now( ) method there is another way to create a time date object i.e. the "**of**" method. This keyword of allows us to create a date or time by passing our own parameters. Note that we always separate parameters with commas.

A reminder to import the relevant libraries.

```
import java.time.LocalTime;
import java.time.LocalDate;
```

Examples:

Time: Hours, minutes, and seconds – this is the parameter order. Parameter values must be logical.

```
LocalTime theTime = LocalTime.of(8,0,0); // This is 08:00am

LocalTime theTime = LocalTime.of(18,20,0); // This is 6:20pm

LocalTime theTime = LocalTime.of(0,0,1); // One 1 second past midnight

LocalTime theTime = LocalTime.of(24,0,1); // Not logical - Error in hour
```

Date: year, month, day – this is the parameter order. Parameter values must be logical.

```
LocalDate theDate = LocalDate.of(2012,12,12); // 12 Dec 2012

LocalDate theDate = LocalDate.of(2,12,12); // 12 Dec 0002

LocalDate theDate = LocalDate.of(2012,30,12); // Error in month

LocalDate theDate = LocalDate.of(12,12,2012); // Error in day
```

DATE AND TIME – our first three date time methods.

It is useful to know which date or time came first. For this we have three methods.

- **isBefore** – returns Boolean when the first date/time comes before the second.
- **isAfter** – returns Boolean when the first date/time comes after the second.
- **isEqual** – returns Boolean when the first date/time is the same as the second.

The login program that follows has two conditions that must be met - therefore the use of the logical AND operator.

1. The login name must match.
2. People may not login before 6am in the morning.

To save space the UI classes will not generally be shown in future unless relevant.

```
// Name and time conditions must both be met.
// People may not login before 6am in the morning

import javax.swing.JOptionPane;
import java.time.LocalTime;

public class LoginConditionalManager
{
   // Global variables for all methods to use
   private String myName = "steve";
   private String name = null;
   private LocalTime theTime = null;
   private LocalTime validTime = LocalTime.of(6,00,00); // no access before 6am

   public void enterName()
   {
            name = JOptionPane.showInputDialog(null, "Enter your name");
            name = name.toLowerCase();
   }

   public void login()
   {
      if(name.equals(myName) && theTime.isAfter(validTime))
      {
         System.out.println("Access granted");
      }
      else
      {
         System.out.println("Access denied");
         System.out.println("Name or time conditions not met");
      }

   }
}
```

Exercise 1.5C.

Using the program above as a template, modify your solution to exercise 1.5B (loyalty points). The 1.5C version must not allow any person to get any discount before 8:30 in the morning.

Exercise 1.5D.

Using the program above as a template, modify your solution to exercise 1.5C (loyalty points). The 1.5D version must not allow any person to get a discount before 8:30 in the morning and after 6:45 in the evening.