**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming 1**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Practical 15 – User-defined Classes and Objects**

This lab sheet moves on to the last topic we shall cover in the module – **user-defined** **classes** and **objects**. Although the module has made many references to the notions of class, object and object reference as we have gone on, it has always been in association with pre-defined Java API classes such as System, JOptionPane, String and so on. Our last two lab sheets will investigate how **user-defined classes** are designed and coded. This will introduce you to some of the very **basics of object-oriented programming** and give you a feel for what 2nd year programming will involve.

**Revision – “Ordinary” Variables and Object References**

In an earlier lab sheet we took a look at “ordinary” variables and object references and how they differ from each other. Recall that an **“ordinary” variable** is a variable associated with a **primitive data type** such as int, float, double, char, boolean etc. For example in the line of code:

int x = 45;

x is declared to be an **ordinary variable** of type int and the value 45 is assigned to the variable x.

Now consider the line of code:

String name = new String(“Jack”);

Here name is declared to be an **object reference** (also called a **reference variable)** of type String) and the String object holding the word “Jack” is assigned to the object reference.

Jack

1330

1330

name

The object

The object reference

An “**ordinary” variable takes up a set amount of memory space** and **a value is stored within that memory space**.

An **object reference takes up a certain amount of memory space** also but the **object it is referencing is not stored within that memory space**. Instead, the **object reference just stores the memory address of the object** it is “pointing” to within its memory space.

In the example above, that memory address begins at location 1330.

**Revision – the new keyword**

As indicated in the example above, the **new** keyword is **synonymous with object creation** in Java. Whenever you see the new keyword in a line of code, then you know an object is being created (and memory space is being allocated to store that object).

Think of all the times recently when you have created, for example, JButton, JFrame and JLabel objects. Each time the line of code involved the new keyword.

**Classes, Objects and Instantiation**

Whenever you **create an object** using the new keyword, you are said to be **creating an instance of the class** it is associated with.

So, when you create a String object using the code:

String name = new String(“Jack”);

You are **creating an instance of the class** String. This process is called **instantiation** in Object-Oriented Programming terminology.

A **class** is often defined to be a **blueprint** for a group of related objects. It is similar to using a blueprint for building houses. You can create many houses, all of the same type from a given blueprint. In much the same way, you can create many objects, all of the same type from a given class. In OOP the class is the **basic building-block** for application development.

However, you **generally need to create (instantiate) objects** from classes in order for the classes to be of use.

**Revision - Pre-defined Classes and Methods**

We have used a multitude of pre-defined classes and their associated methods throughout the module. Recall that in Java, there is a definite organisation to the API. There are the **packages** at the top, each of these **contain many related** **classes** and each of these classes in turn typically **contain many related methods**.

Whenever we needed to use a particular method, we always needed to know the class it belonged to and also the package in which the class was defined. To actually make the method call, we also needed to know the method name, the number, order and type of its arguments as well as its return type.

Finally, we needed to know whether or not the method was a **static** or an instance **method**.

**Revision – Static and Instance Methods**

If a method is **static** then we just **call the method directly on the class it belongs to** as follows:

text=JOptionPane.showInputDialog(“This is a static method”);

The method we have used so often, showInputDialog() is a static method. Therefore we **should call it directly on the class it belongs to** rather than on an instance of that class. An object of the class does not need to exist in advance of calling a static method. In other words we shouldn’t really do this:

JOptionPane dialogRef = new JOptionPane();

text=dialogRef.showInputDialog(“This is a static method”);

because now we have created an instance of JOptionPane() – using the new keyword – and then called the method on that instance. This code is still actually valid but **not conventional** since most professionals never create an instance when calling a static method.

There have been numerous examples of **instance methods** in the GUI section of the module. The code:

jFrameWindow.setLayout(flowLayout);

jFrameWindow.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

shows 2 examples of instance methods called setLayout() and setDefaultCloseOperation().

These are instance methods rather than static methods because **an instance of (an object of) JFrame must exist before they can be called**. The instance in this case is being referred to by the object reference **jFrameWindow**.

Interestingly though, see that the setDefaultCloseOperation() takes as an argument **JFrame.EXIT\_ON\_CLOSE**. This is a **constant** field (how can I tell this?) defined within the JFrame class but it is **static**, because we are **able to refer to it directly on its class**.

Note that whenever a method is called on either a class or an instance of a class, a **dot** must separate the two.

In this module we never went into the specifics about *why* some methods are static and some are instance methods. This will be explained in your OOP2 module. For now, it is good enough that you are aware of the two types of method and can call each type successfully when needed.

**Object-oriented and Procedural Programming**

For most of the module, we had just the one class in each program and within this class we had the main() method. Then we popped our actual code into the main() method and that was it.

This type of coding layout is referred to as **procedural programming** and (except for the class definition) is similar to the kind of layout you see in procedural programming languages such as C, Pascal and FORTRAN.

As mentioned last term, there are **two main programming styles** in existence – procedural programming and object-oriented programming. The **concept of a class does not exist in procedural languages** and this is the main difference between the two styles. As a consequence, you cannot create objects in these languages either, nor can you call methods on instances of a class. The structure is very different to the object-oriented programming language setup and there are many useful OO features completely missing from the procedural programming style – the equivalent of methods still exist in procedural languages but they are not defined within classes as they are in OO languages.

**User-Defined Class Structure**

You have already created many user-defined classes in this module – in fact, every Java application you wrote involved at least one user-defined class, this being the public class you named your Java file after.

**public class ClassName {**

You recall that class is a **keyword** and that it **defines a structure** within a Java program.

For example, a partial **definition of the Java API Math class** looks as follows:

:

:

/\*\*

\* The class <code>Math</code> contains methods for performing basic

\* numeric operations such as the elementary exponential, logarithm,

\* square root, and trigonometric functions.

\*

\* other comments omitted for brevity here \*/

public final class Math {

:

:

public static final double E = 2.7182818284590452354;

public static final double PI = 3.14159265358979323846;

public static double sin(double a) {

//code to find sine of an angle in radians

}

public static double cos(double a) {

//code to find cosine of an angle in radians

}

public static double tan(double a) {

//code to find tangent of an angle in radians

}

:

:

:

}

It is useful to see what pre-defined classes look like before attempting to create your own user-defined classes.

Within the class definition, you can see that the Math class defines two **attributes** (fields) called E and PI. Hopefully you can see that these attributes are **constants** because of the keyword **final** being used. They are also **static**, which means that when we re-use the Math class in our own programs, we can get the constants PI and E through the code

**Math.PI** and

**Math.E**

as discussed recently.

It also has various methods defined, many of which you have used before. The partial definition here shows the definition headers for the sin(), cos() and tan() methods but there are many more as you know.

So you can see that **a class usually contains various parts such as constants and methods**. However, a **class can also contain regular variables** (both **ordinary and object reference**) and **even other classes**.

Just think back now to the GUI programs you wrote recently. There were typically several “global” variables/references defined e.g.

public class Exercise4 {

JLabel titleLabel,firstNameLabel,surNameLabel;

JComboBox jComboBox;

JTextField firstNameField, surNameField;

String titles[];

String title;

public Exercise4()

{

JFrame jFrameWindow = new JFrame("Enter Details");

FlowLayout flowLayout = new FlowLayout();

jFrameWindow.setLayout(flowLayout);

The GUI application class here defines 8 “global” object references. These object references were made “global” because they needed to be accessible from the event-handling method(s) defined within the application.

In OOP terminology, these “global” variables/object references are called **attributes** of the class (just as E and PI are attributes of the Math class). Attributes of a class are **defined outside any particular method of the class** and are therefore **“visible” to all methods defined within the class**. They either store data or a memory address (if a reference variable)

The GUI application always had a main() also, as a means of kick-starting the whole GUI creation process.

Then there was normally a **private inner class** defined within the public class. This contained a method which was capable of handling some kind of event that occurred on a particular GUI component at runtime.

So, in general , a **public class definition** has the following appearance

**public** **class** *ClassName* **{**

*attribute declarations*

*method definitions*

*private class definitions*

**}**

Any particular class definition does not need to contain attributes, methods or private classes. However, almost all public class definitions contain at least one method definition and often have many more than this. Also most classes contain a few attributes (some contain none while others contain lots) while relatively few contain private inner classes.

**Design of a User-Defined Class**

In this first example of a user-defined class, we look at defining a class called **Thermometer** to represent the temperature (current, maximum and minimum) for the day. We also assume that temperature is represented as an **integer**. We wish to perform the following operations in relation to the temperature:

● Set the current temperature

● Get the current temperature

● Get the maximum temperature

● Get the minimum temperature

● Display the current, maximum and minimum temperatures

Each of these operations will be **carried out by an individual method** and we just need to come up with the algorithms for defining these methods.

There will be **3 attributes** in this class – **currentTemp**, **maxTemp** and **minTemp**.

Up until now, whenever you have declared “global” variables/object references i.e. **attributes**, you did not specify whether or not they could be accessed outside the class they were defined in. This is because up until now this was never a consideration as the classes you wrote were never really going to be re-used in the OO sense of the term. Now, as we move on to write our own, re-usable, classes this is a vital consideration.

It is **normal to define the attributes of a class as private** (even though you saw earlier that the constant attributes defined in the Math class were actually public). This means that the **attributes cannot be accessed directly outside of the class they are defined in** – the exact reasons behind this will be explored in more detail in your OOP2/OOP3 modules. For now you can just take it that, whenever you create a class attribute, it should be declared private (unless otherwise specified).

So here, the 3 attributes of the class will be declared as follows:

private int currentTemp;

private int maxTemp;

private int minTemp;

remember that these will be **declared outside of any method** as they are attributes of the class.

Likewise, we need to decide whether the methods we will define are going to be private or public. We should make the method **public if its needs to be accessible outside of the class it has been defined in**. This is the case for the vast majority of methods you will encounter.

All the 5 methods specified earlier for this class will be public in nature since the user of the class needs to be able to carry out all of the operations specified by calling the appropriate public method.

In this case, **we will make all the attributes and methods of the class** **non-static**. Therefore we have **3 instance attributes** and **5 instance methods** within the class definition. Again, the reasons behind choosing to make attributes/methods static or instance will be examined in OOP2/OOP3. For now, you just need to be able to apply the keyword where specified and use static attributes/methods correctly.

The methods themselves will therefore have the following definition headers

● **public void setCurrentTemperature(int cTemp)**

This will set the current temperature to the temperature specified by the user (passed in as an argument to the method)

● **public int getCurrentTemperature()**

This will get (return) the current temperature

● **public int getMaxTemperature()**

This will get (return) the maximum temperature

● **public int getMinTemperature()**

This will get (return) the minimum temperature

● **public String toString()**

This will return the values of the 3 attributes of the class in String form for display purposes.

These are the 5 public methods we started out to code. However, we **need an additional method**. When the user sets the day’s current temperature, depending on the value of the current temperature supplied, the program may need to update the values of the maximum and minimum temperatures. This extra method will look as follows:

● **private void updateMaxMinTemp()**

This private method, which is called by the method setCurrentTemperature(), will perform tests to determine whether the value of the attribute currentTemp is below the minimum temperature or above the maximum temperature. These values are directly available through the attributes minTemp and maxTemp here but could also be accessed indirectly through the methods getMaxTemperature() and getMinTemperature().

**Constructors**

In addition to the methods necessary to implement operations, every class can have one or more of a **special type of method** called a **constructor**. A constructor always has an **identical name to the class** it is defined in. The appropriate constructor (as specified by the argument list) **executes automatically when an instance of a class is created**. Constructors are used primarily to **ensure that the attributes of a class are initialised** with suitable values.

You **have been using constructors throughout the module**. Any time you created an object, in fact, you made a call to a constructor e.g.

JFrame jFrameWindow = new JFrame("Enter Details");

Here the JFrame constructor is being called and the String argument “Enter Details” is passed to it to effectively initialise the title bar of the JFrame window with the text shown. Making calls to constructors is exactly the same as making calls to regular methods, except that the **call is always preceded by the new keyword**.

One thing that sets a constructor apart from a regular method is that it has **no return type at all** (not even void). This means that **constructors can never return a value** to a calling method. This makes sense though because constructors are only meant to set up attributes with values, not return values. A call to a constructor will implicitly return the memory address of the newly created object, which may be stored in an object reference variable.

Every class that you examined in the Java API has a number of constructors – they are **differentiated from each other through their argument lists**.

For our Thermometer class here, we will include 2 constructors. These will be a **no-argument constructor**, which will initialise the values of all 3 attributes to so-called “default” values and a 1-argument constructor that will initialise the attributes to a value specified by the user in the argument.

The method definition headers for these constructors will be:

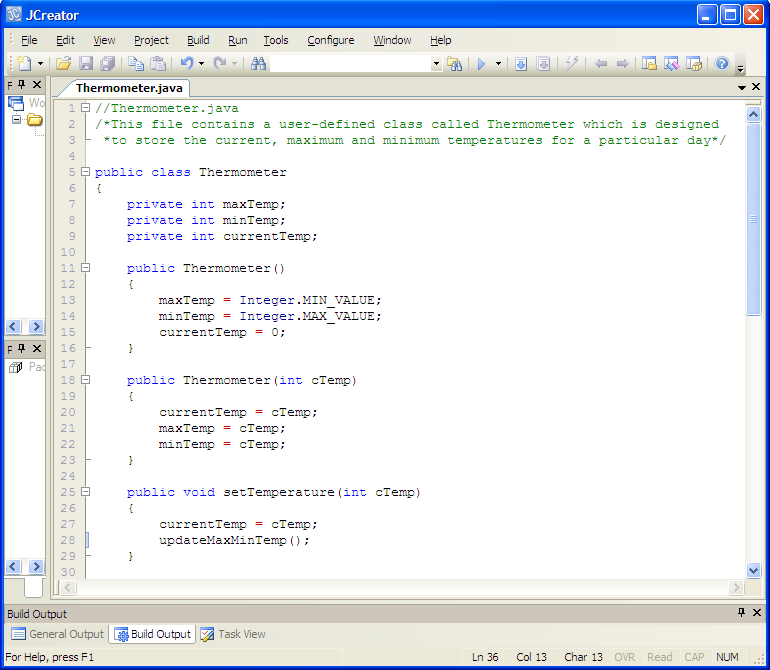
● public Thermometer()

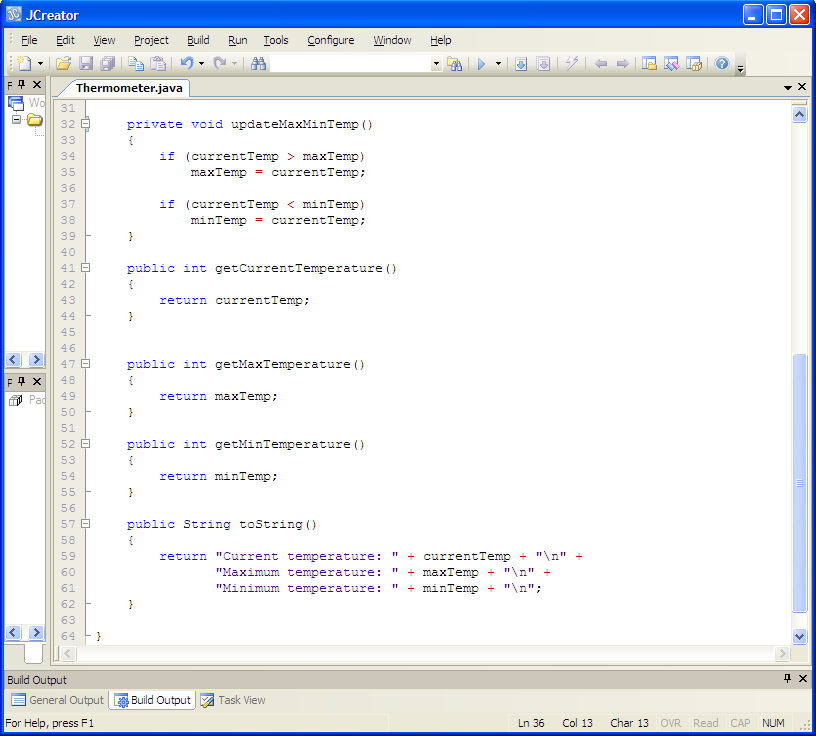
and

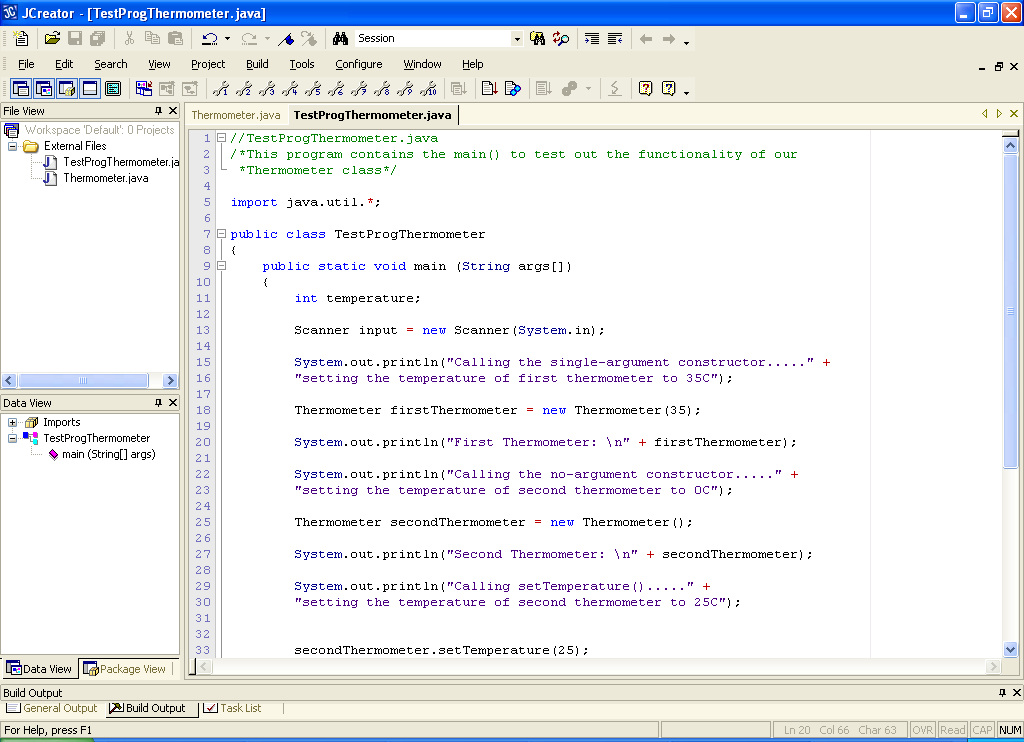
● public Thermometer(int cTemp)

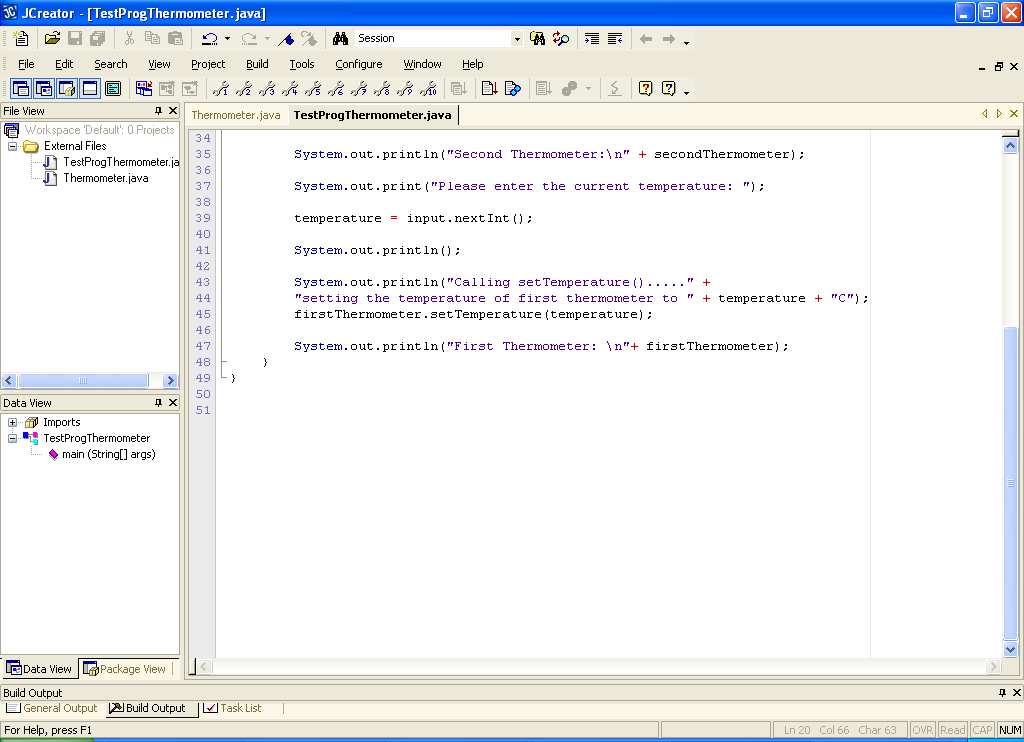
**Aim**: To demonstrate a user-defined class called **Thermometer** which can be used to set the current temperature as well as keep track of the maximum and minimum temperatures. These values can also be retrieved and displayed.

**Java Code:**









**Analysis of the Thermometer class:**

● The first thing to note about the program above is that it is split between **2 separate Java files** called **Thermometer.java** and **TestProgThermometer.java**. This is the nature of object-oriented applications – there are generally several Java files within the one “overall” application. However, no matter how many files there are in an application, **only the one main() method can be run**. In this case, the main() is within the TestProgThermometer.java file.

● Starting off with the Thermometer class, we see that it declares **3 private** **attributes** as specified earlier. Then there is a **no-argument constructor**. This contains the code:

public Thermometer()

{

maxTemp = Integer.MIN\_VALUE;

minTemp = Integer.MAX\_VALUE;

currentTemp = 0;

}

Remember that a **constructor has no return type**. Also remember that it **must have the same name as the class** in which it is defined. The idea of the constructor is to **initialise the attributes** with suitable values.

In this case the value of the maxTemp attribute is set to the value **Integer.MIN\_VALUE** – this is the smallest possible integer value allowed in Java (its around -2 Billion). The value of the minTemp attribute is set to **Integer.MAX\_VALUE** – this is the largest possible integer value allowed in Java (its around 2 Billion). The reason for setting the max and min temperature values to these strange looking values may not seem obvious.

The idea here is really to ensure that when the user eventually supplies a temperature for the current temperature attribute using the setTemperature() method for the first time, that the maximum and minimum temperature will get set to the correct value. This happens as a result of the code contained within the updateMaxMinTemp() method.

● The 1-argument constructor contains the code

public Thermometer(int cTemp)

{

currentTemp = cTemp;

maxTemp = cTemp;

minTemp = cTemp;

}

This takes the user-supplied argument cTemp and sets the attribute currentTemp to this temperature value. It then sets both maxTemp and minTemp to this particular value also. Effectively, when this constructor is called, all 3 temperature attributes will be the same, at least momentarily.

● The next method is

public void setTemperature(int cTemp)

{

currentTemp = cTemp;

updateMaxMinTemp();

}

In object-oriented programming terminology this method is called a **mutator** because it **mutates (changes) the value(s) of a certain attribute(s) of the class**. This one changes the value of the currentTemp attribute and may also change the value of either maxTemp or minTemp depending on the value of the argument cTemp. In OOP you can always identify a mutator method because, by convention, they **begin with the word “set”**.

● The next method is

private void updateMaxMinTemp()

{

if (currentTemp > maxTemp)

maxTemp = currentTemp;

if (currentTemp < minTemp)

minTemp = currentTemp;

}

This method is defined as a **private** **method** – therefore it **cannot be directly accessed outside of the Thermometer class** – an attempt to do so would cause a **syntax error**. This private method just updates the values of the attributes maxTemp and minTemp as necessary, depending on the value of currentTemp. It is called by the public method setTemperature() above. private methods are often referred to as **utility methods** because they are normally used by other methods within their class to carry out some task. They serve no useful purpose in isolation because they **cannot be called directly on an instance of the class**.

● The next couple of methods are:

public int getCurrentTemperature()

{

return currentTemp;

}

public int getMaxTemperature()

{

return maxTemp;

}

public int getMinTemperature()

{

return minTemp;

}

In object-oriented programming terminology these methods are called **accessors** because they **allow you to** **access** **the value of a certain attribute of the class**. These allow us to retrieve the values stored in the attributes currentTemp, maxTemp and minTemp. You can always identify an accessor method in OOP because, by convention, they **begin with the word “get”.**

● The last method defined within the class Thermometer is:

public String toString()

{

return "Current temperature: " + currentTemp + "\n" +

"Maximum temperature: " + maxTemp + "\n" +

"Minimum temperature: " + minTemp + "\n";

}

This method **returns** **a String that gives the current state of the objects attributes**. By convention, there is normally a toString() method defined in every class and its purpose is **to return the current state of the object it is called on** in the form of a String.

Recall that whenever you needed to display the value of an object to the screen, you either used System.out.println() or JOptionPane.showMessageDialog() or, more recently, a JLabel. In all these cases, when it comes to displaying information, a String is expected. When you create a class of your own and create objects from the class, each object will have its own state but it will not always be in String form e.g. our class Thermometer has 3 attributes that define its state but all 3 are integers.

If we were to create a Thermometer class without defining a toString() method, then create a Thermometer object and refer to it by the reference therm1 and try to display its state using the following code:

System.out.println(“The state of the thermometer object is ” + therm1);

Then the code is actually valid and compiles but at runtime we will get something similar to the following:

**The state of the thermometer object is Thermometer@11b86e7**

So the state of the Thermometer object looks non-sensical and has nothing to do with the actual data values contained in the object. This is because the **Java runtime will provide and call a default toString() method on an object automatically** within a println() if no other method is called on it, as is the case here. The default toString() method will display the non-sensical output we see above. Therefore we have two choices - write our own toString() method to replace the default one or call the method something else and make sure we call it whenever it is needed. It is good to stick with **convention** though and therefore, we are better to always define a toString() method for our class should we need to know the state of any objects created from it (which is usually very handy e.g. when testing out a class)

● Note that this class does not contain a main() method. This is the first time you have seen this during the module, but **this is the norm for object-oriented programs**. Most classes for an OO application will be designed along the lines of the Thermometer class, with no main(). We look to identify the **data storage parts** (the attributes), then we look to identify the **functionality** the class will be capable of (the methods) and then we write the class. We could have put a main() method into this class and not bother with the TestProgThermometer.java file at all but this would be going against the OO “grain” where the idea is always to **create** **classes that are completely self-contained and re-usable** – sticking a main() into this class would not be the end of the world but there would be no point in calling the main() method from another class – it would not make logical sense.

Therefore the main() is placed into a separate **“driver” class** and the express purpose of the driver class is to **test out the functionality** of the user-defined Thermometer class to ensure it is operating correctly.

**Analysis of the TestProgThermometer class**

● The driver class contains the main() and it begins by creating a Scanner object to allow reading from the keyboard. So this will be a console application.

● Some of the println() method calls in the driver are there for explanation purposes - they are just there for your benefit so that you can hopefully make sense of the output.

● The code

Thermometer firstThermometer = new Thermometer(35);

creates a Thermometer object using **new** and a call to the **1-argument constructor**. The argument 35 sets the value of the attribute currentTemp to 35. It would also reset maxTemp to 35.

The println() following this displays the current state of this Thermometer object. Note that the call to the toString() method in the println() is implicit, but you may, of course, put it in if you prefer.

● A second Thermometer object is created through the **no-argument constructor**. This will set currentTemp to 0 as well as setting minTemp and maxTemp to values of roughly 2 Billion and -2 Billion respectively.

● After this, the current temperature of the second Thermometer object is set to 25C through a call to the **mutator** method **setTemperature**(). This may also affect either the value of minTemp or maxTemp depending on their current values.

● Next, the user is prompted to enter a temperature value and this is read in and used to set the temperature of the first Thermometer. Again, this may affect the minTemp or maxTemp attribute value.

**Organising your Work**

You should have a folder under X: called OOP1Stuff created. This time, create a folder called **Lab15** within OOP1Stuff to save your work from this lab session.

**Typing in Code for the Program Just Analysed**

Click the **New File** icon on the JCreator IDE and save the first file as **Thermometer.java** and the second as **TestProgThermometer.java** in your Lab15 folder. Now, in order to get used to the layout of an OO style application, type in the entire code for the program above. It’s a pain but it’s the only way to get used to this new style of coding. It won’t be wasted as it is the way you will be laying your code out from now on, through the remainder of your OOP modules in second and third year.

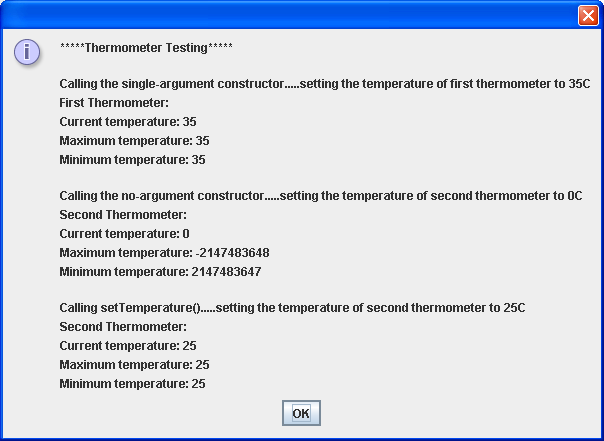
If your program has any errors or warnings, have a look at the edit window and check to ensure that the code is exactly as indicated earlier, including all **semicolons** (**;**) and concatenation operators (+) and ensuring that letters are written in lowercase where indicated. If you spot any differences correct them and compile again until the program is syntax error-free.

Once you are free from errors, run the program and test it fully. You will see that the user input is **not validated** at all.

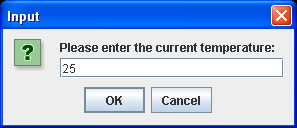
**Exercise 1**

Save the driver class as **Exercise1.java** and then modify it so that it becomes a GUI application rather than a console one.

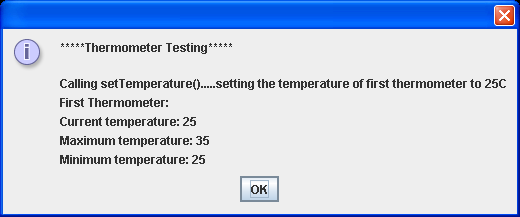
Your application will run in a similar fashion to the following screenshot



**After the user clicks the OK button**



**After the user clicks the OK button**



**Exercise 2**

You may have noticed that the **driver class never actually tests any of the accessor methods which is a major flaw**. The easiest way to ensure these are tested is to modify the existing **toString**() method in the Thermometer class. Modify this method now to have them refer to the accessors rather than referring to the attributes directly. Save the modified class as **Exercise2.java**. Note that you will also need to modify the existing driver class to account for the different class name. Save this as **TestExercise2.java**. The driver should run exactly the same as before.