**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming 1**

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

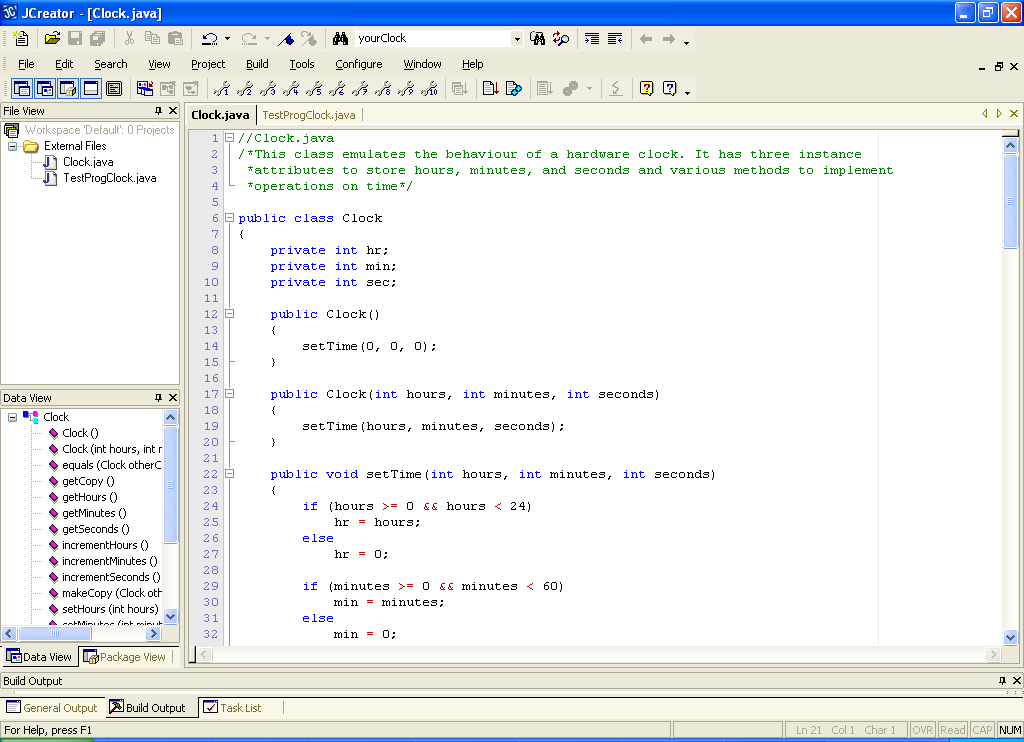
**Practical 16 – User-defined Classes and Objects**

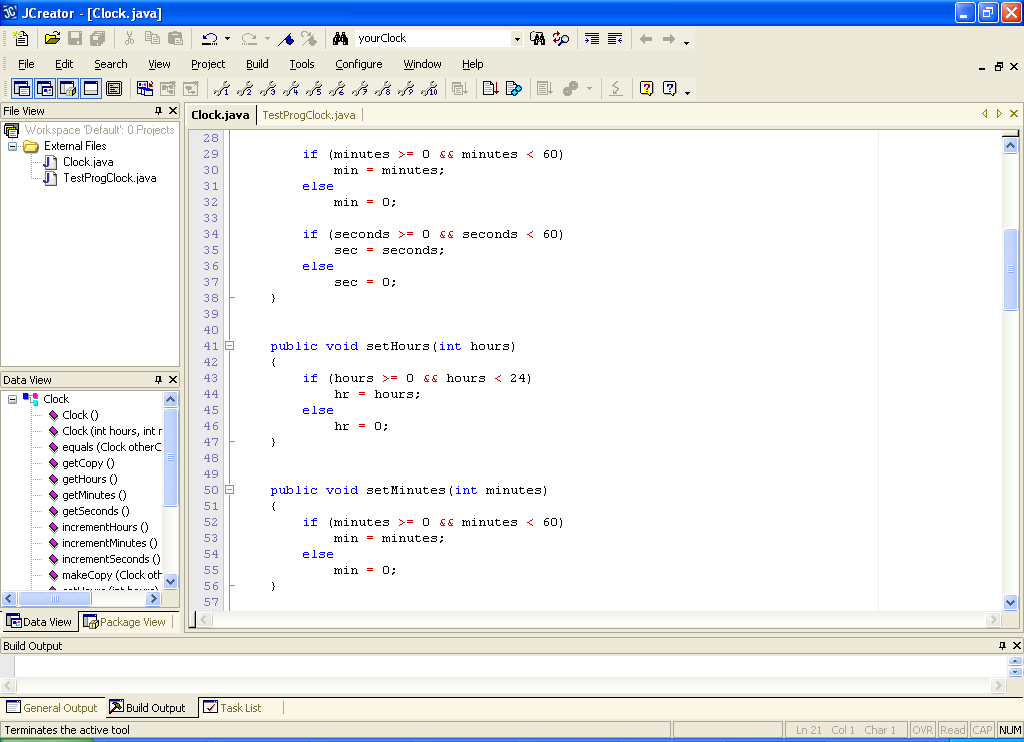
The last lab sheet introduced many basic OO concepts. Hopefully you can recall what OO terms like class, object, object reference, accessor, mutator, constructor, public and private refer to since these terms will be used when specifying OO problems. This lab sheet looks at some more user-defined classes, their corresponding driver classes and also introduces the OO documentation standard called the **Unified Modeling Language** (UML).

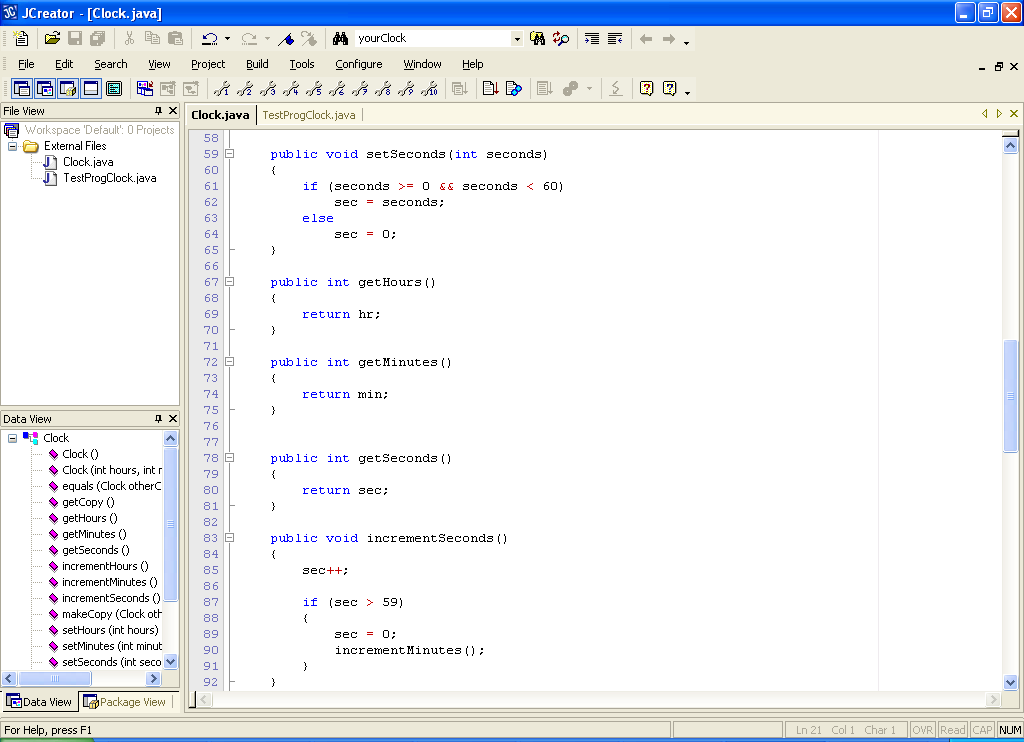
**Aim:** This program shows a user-defined class called **Clock** which can be used to tell the time of day, as well as set the time (note that our version here has nothing to do with the computer’s system time however).

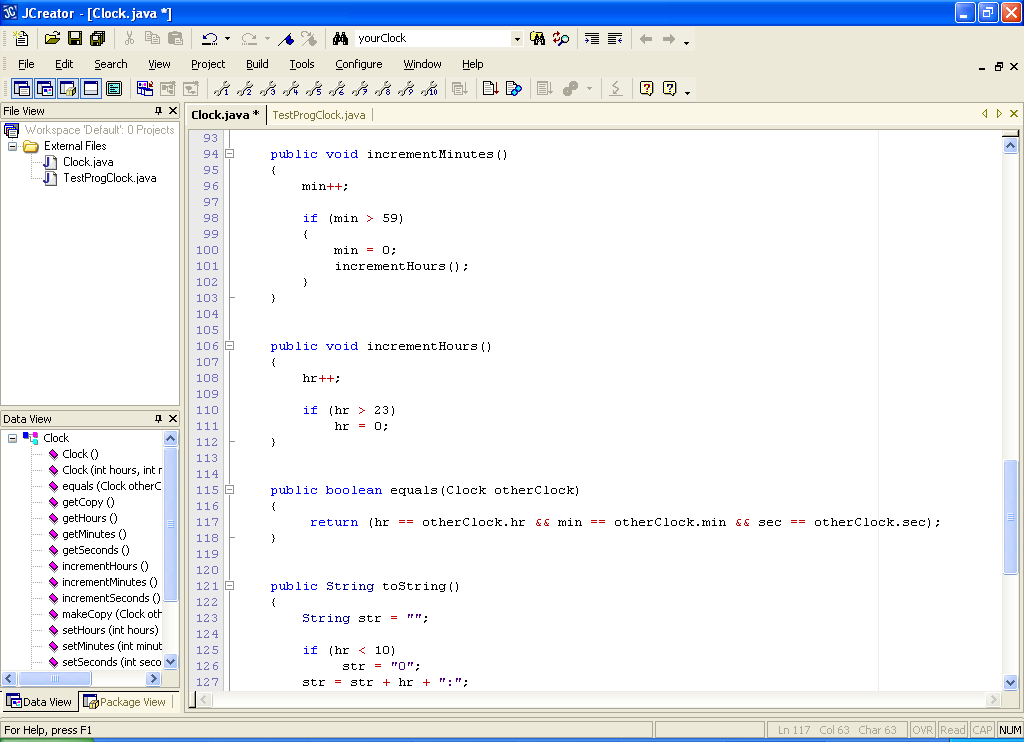
**Java Code:**

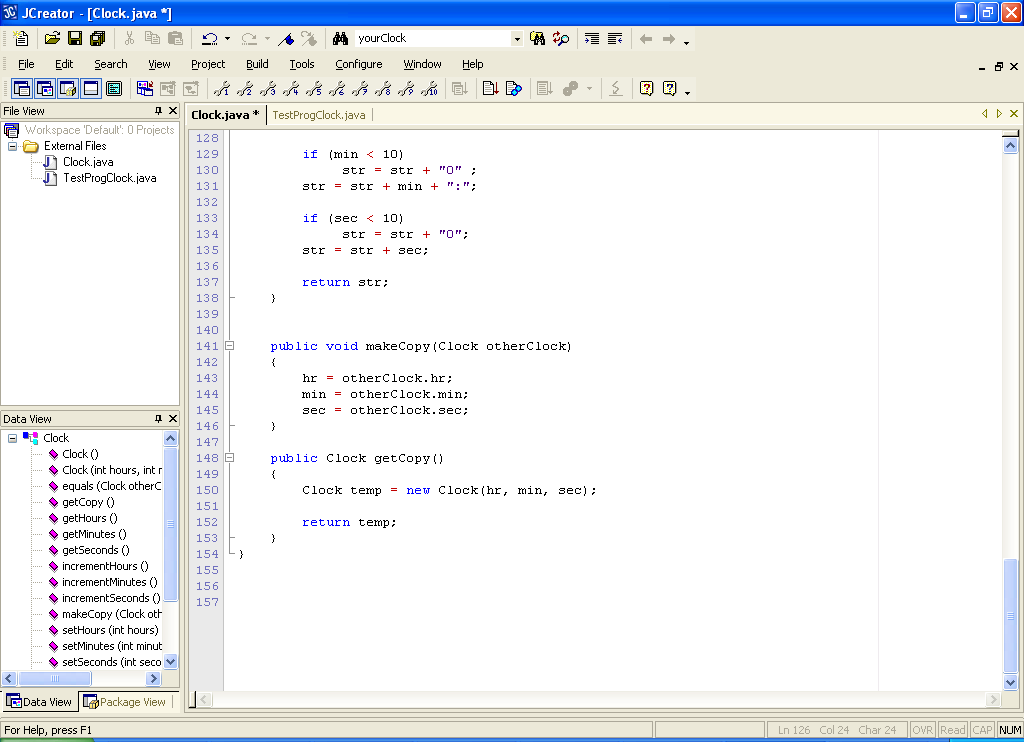
**The Clock class**



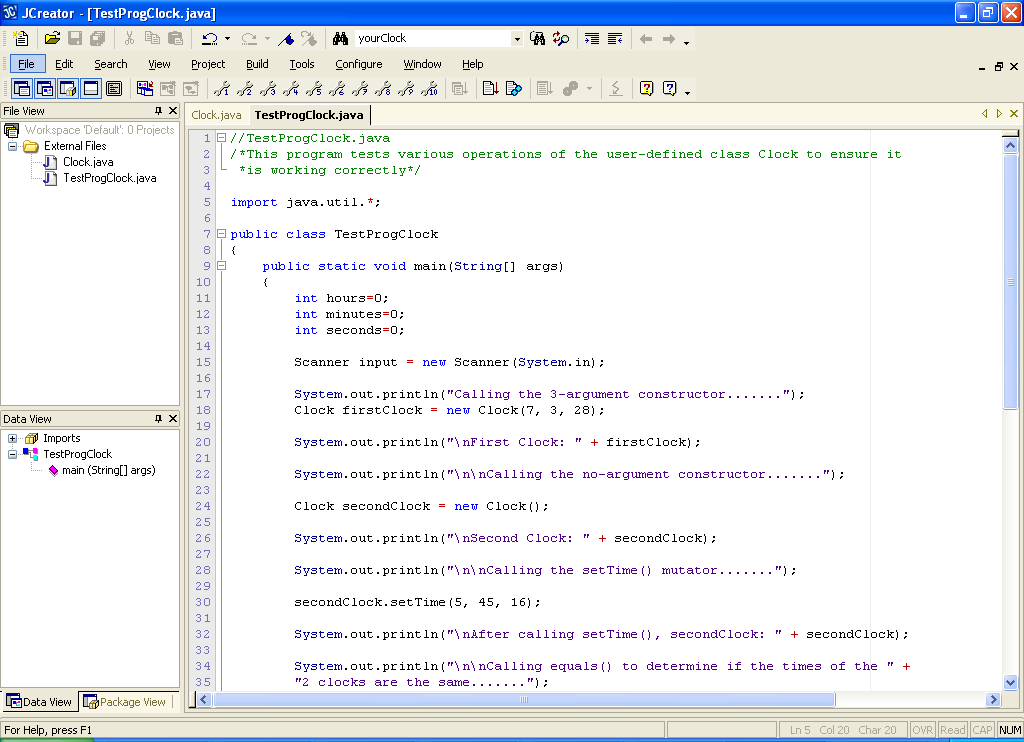


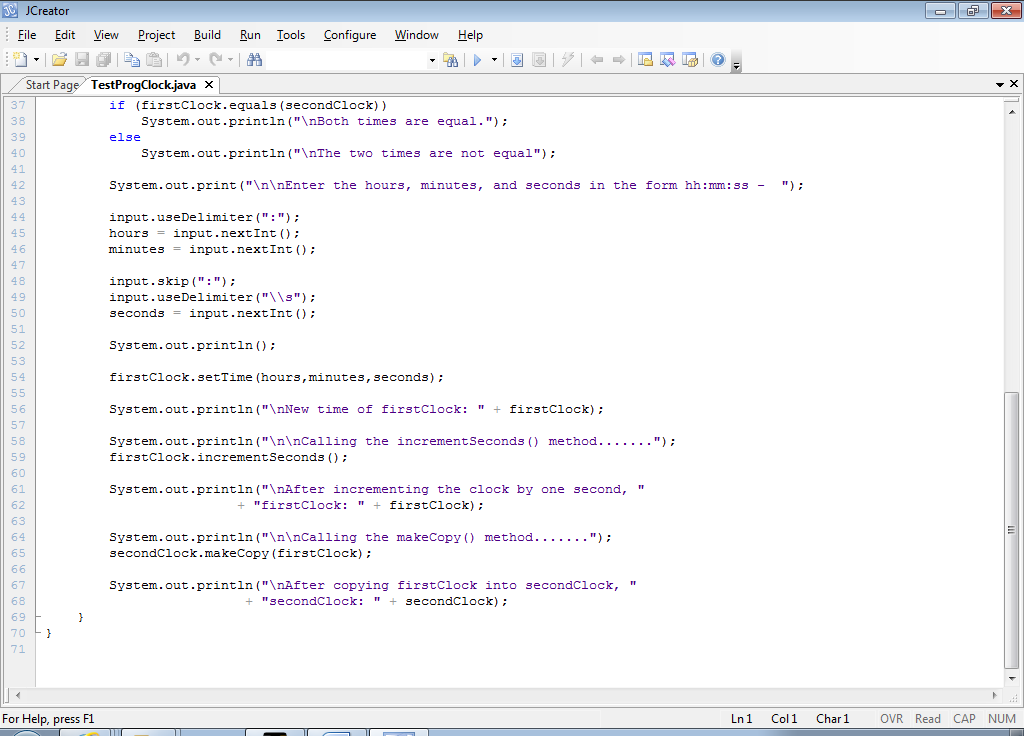






**The driver class TestProgClock.java**





**Analysis of the Clock Class:**

● The Clock class needs to represent a hardware clock. Therefore it needs to be able to store hours, minutes and seconds so the user can tell the time. These are all declared as integer attributes of the Clock class. They are declared **private** also, preventing them from being directly accessed outside of the Clock class.

● The class contains a **no-argument constructor** as follows:

public Clock()

{

setTime(0, 0, 0);

}

Like all constructors, this is used to initialise the attributes of instances of the class with a set of suitable initial values. In this case, the no-argument constructor gives all the attributes the value zero, effectively setting the time of the Clock object to 00:00:00.

Notice here how the constructor makes a call to another method within the Clock class, setTime() in order to initialise a Clock object. This is just another example of **code re-use**.

● Next there is a 3-argument constructor:

public Clock(int hours, int minutes, int seconds)

{

setTime(hours, minutes, seconds);

}

This one takes 3 arguments that represent user-supplied values and set the time of the Clock to that particular value. Again, it calls the setTime() method to actually do the setting of the hr, min and sec attributes.

● Next comes the setTime() method itself:

public void setTime(int hours, int minutes, int seconds)

{

if (hours >= 0 && hours < 24)

hr = hours;

else

hr = 0;

if (minutes >= 0 && minutes < 60)

min = minutes;

else

min = 0;

if (seconds >= 0 && seconds < 60)

sec = seconds;

else

sec = 0;

}

This method takes 3 arguments representing the hours, minutes and seconds desired and then performs some basic validation on their values to ensure they lie within the appropriate ranges. If any of them fail, the corresponding attribute is then set to a “safe” value of zero.

It is highly recommended to put validation code into the **mutator methods** such as setTime() to **ensure that the object is always in a consistent state**. Otherwise, it would easily be possible to set the attributes to undesirable values which could ultimately lead to logical errors or runtime errors.

● The next 3 methods are also **mutators**. The only difference is that these ones set specific attributes whereas setTime() sets all 3 in one go. Notice that the validation code is exactly the same as that used in the setTime() method.

public void setHours(int hours)

{

if (hours >= 0 && hours < 24)

hr = hours;

else

hr = 0;

}

public void setMinutes(int minutes)

{

if (minutes >= 0 && minutes < 60)

min = minutes;

else

min = 0;

}

public void setSeconds(int seconds)

{

if (seconds >= 0 && seconds < 60)

sec = seconds;

else

sec = 0;

}

● The next 3 methods are the **accessor** methods of the class. These simply allow the values of the Clock attributes to be obtained. Hence, their bodies contain just the one line of code, a return statement which refers to the particular attribute concerned.

public int getHours()

{

return hr;

}

public int getMinutes()

{

return min;

}

public int getSeconds()

{

return sec;

}

● The next 3 methods allow the user of the Clock to “manually” increment the hours, minutes and seconds of the clock. Its just like the button on the hardware clock that you keep pressing to set the time manually. There is again some validation involved here to ensure that the values remain consistent. For example, as soon as you reach 59 minutes and then increment again, the min attribute then goes to 0, not 60. Also, this scenario has the knock-on effect of calling the incrementHours() method to increase the value of the hrs attribute by 1 also. This is a further example of **code re-use**.

public void incrementSeconds()

{

sec++;

if (sec > 59)

{

sec = 0;

incrementMinutes();

}

}

public void incrementMinutes()

{

min++;

if (min > 59)

{

min = 0;

incrementHours();

}

}

public void incrementHours()

{

hr++;

if (hr > 23)

hr = 0;

}

● The next method is as follows:

public boolean equals(Clock otherClock)

{

return (hr == otherClock.hr && min == otherClock.min

&& sec == otherClock.sec);

}

This method compares two Clock objects for equality, to see whether both are giving the same time. If they are, then their values for the attributes hr, min and sec must all be the same. Notice how you can compare the hr attribute of one Clock object to another using the code

hr == otherClock.hr

the hr on the right of the assignment statement refers to the hr attribute of the method argument otherClock, whereas the hr on the left of the assignment statement refers to the hr attribute of the object on which the method is actually called. So, it might be called as follows:

clock1.equals(clock2)

if the time of clock1 was 12:34:45 and the time of clock2 was 13:45:56 then the value of hr would be 12 and the value of otherClock.hr would be 13. They would not match and so the equals() method would return false.

It is customary for a class to contain an **equals**() method so that instances of the class can be compared for attribute equality. You have already seen many times the use of the String class’ equals() method to compare 2 String objects for equality.

● The next method of the class is a **toString**() method

public String toString()

{

String str = "";

if (hr < 10)

str = "0";

str = str + hr + ":";

if (min < 10)

str = str + "0" ;

str = str + min + ":";

if (sec < 10)

str = str + "0";

str = str + sec;

return str;

}

This method just returns the time associated with the Clock object in the form hh:mm:ss, the normal way in which a hardware clock tells us the time. Note the way that a check is performed to see whether the hours, minutes or seconds are less than 10. This is done to ensure that, if they are, the value will be prepended with a “padding” of 0. This is to make sure the time appears as it normally would for a hardware clock.

● The next method is

public void makeCopy(Clock otherClock)

{

hr = otherClock.hr;

min = otherClock.min;

sec = otherClock.sec;

}

This method effectively sets the time of the Clock object the method is being called on, to the values of the other clock’s attributes. It effectively synchronises one clock with the time of the other clock it accepts as an argument. It is just like when you set your computer’s clock according to the time of some Internet Time Server. You take it that the server’s time is accurate and you set your computer’s clock accordingly.

So, if the time of clock1 was 12:34:45 and the time of clock2 was 13:45:56 then the code

clock1.makeCopy(clock2);

would then set the time of clock1 to 13:45:56 also.

● The last method is

public Clock getCopy()

{

Clock temp = new Clock(hr, min, sec);

return temp;

}

This method creates a new Clock object and sets its time to the current time of the clock the method is called on. It then returns a reference to this newly created Clock object. It can be used to set the time of a Clock object to the time of an existing clock. It is similar to the makeCopy() method but, in this case, the object on which the method is called does not change at all. Instead its state is effectively copied to a brand new Clock object.

So, if the time of clock1 was 12:34:45 and the time of clock2 was currently 13:45:56 then the code

clock2 = clock1.getCopy();

would set the time of clock2 to 12:34:45 also. Importantly, clock1 and clock 2 would now **refer to different objects** that happen to have the same time.

**Analysis of the TestProgClock Class:**

Have a good look through the code for the driver class now and see if you can make sense of it – with the explanatory println() messages it should be okay. The most difficult part in the code has absolutely nothing to do with testing the Clock class! The code is:

System.out.print("\n\nEnter the hours, minutes, and seconds in the form hh:mm:ss - ");

input.useDelimiter(":");

hours = input.nextInt();

minutes = input.nextInt();

input.skip(":");

input.useDelimiter("\\s");

seconds = input.nextInt();

This code involves using the Scanner class to read in the user’s input in a particular way. The user is prompted for the time in its conventional format and then the delimiter for the Scanner object is set to a colon (by default it is a whitespace character). This splits up the time into hours, minutes and seconds but, unfortunately, the seconds part is not “tokenized” by the Scanner object (if we put another colon after it i.e. hh:mm:ss: then it would be but that is not the traditional time format).

Therefore the scanner makes it as far as the minutes part, which is duly read in by nextInt() and then we are left with the :ss part of the time. The code

input.skip(":");

literally skips over the colon character for us and then the we reset the delimiter back to its default value of whitespace (which is denoted by the **regular expression** \\s). After this, we can then apply the nextInt() method again to pick off the seconds part of the number and we are done – phew! You don’t need to recall any of this part for the exam though.

**Organising your Work**

You should have a folder under X: called OOP1Stuff created. This time, create a folder called **Lab16** within JavaStuff 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 **Clock.java** and the second as **TestProgClock.java** in your Lab16 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 tough going but it’s the only way to get used to this new style of coding.

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 properly validated** at all.

**Exercise 1**

Save the Clock file as **Clock2.java** and the driver file as **TestProgClock2.java**

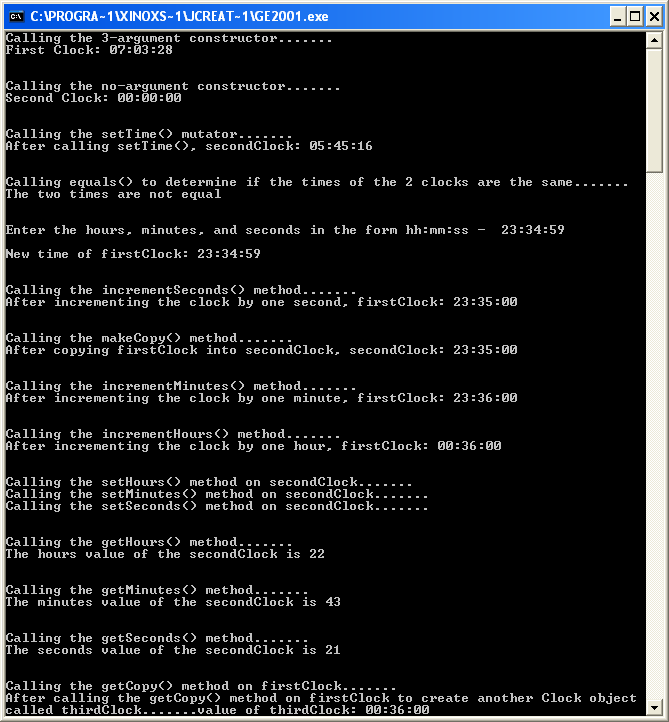
It was mentioned during the analysis of the program that the code in the setHours(), setMinutes() and setSeconds() methods is replicated in the setTime() method. There is nothing wrong with that, of course, but an opportunity for **code re-use** has been missed out on here. You should now modify the setTime() method so that it reuses the setHours(), setMinutes() and setSeconds() methods instead.

When you see the method after this, you will realise just how handy code re-use can be.

**Exercise 2**

Save the **TestProgClock2.java** file as **TestProgClock3.java** and the file **Clock2.java** as **Clock3.java**.

The TestProgClock2 driver class only tests a certain amount of the functionality of the Clock2 class. Make the necessary changes to the classes so that the driver fully tests the class. Your program should then run as indicated below:



**Unified Modeling Language (UML)**

**UML** is the **most widely used graphical representation scheme for modeling object-oriented systems**. Many software system designers use the language (in the form of various kinds of diagrams) to model their systems.

Many programmers starting off just sit down and type code. This is fine if the problem domain is very small and you can work out the logic in your head without too much trouble. This depends on how good you are at working problems out in your head, of course! However, this system normally only works for programs that are relatively simple in terms of the **algorithm** used and are normally no more than 100 lines of code or so.

Then there are some problems that are **a little more complex**, either in terms of the number of tasks to be performed by the program or just in terms of the complexity of the algorithms involved. For these kinds of problems, it is very useful to write a **pseudocode** solution to the problem **before even going near the code editor**. We have applied this technique in last terms module in the problem-solving section and hopefully some of you have also found it to be useful for some of the trickier problems this term. Still, the pseudocode approach is normally limited to a few hundred lines of code at most, meaning it is perfectly adequate for everything we have done in first year.

Then there are the **complex problems** that typically involve a large number of tasks and possibly several complex algorithms. In these cases, the best approach to take is to follow a detailed process for **analyzing** your projects requirements i.e. ***what* the system is supposed to do** and then developing a **design** that satisfies them i.e. **deciding *how* the system should do it**. Ideally, this **should be done before writing a single line of code**. If this process involves analyzing and designing the system from an object-oriented point of view, it is called an **Object-Oriented Analysis and Design** (**OOAD**) process.

Experience has shown that this process **can save valuable time, energy and money**, preventing projects from “falling apart” half way through the coding stage because of a badly planned (or just no) design.

UML is the most commonly used language today for communicating the results of any OOAD process and you will see and use it many times in the next few years to document programs and projects.

**UML Class Diagrams**

When it comes to developing OO applications, the approach should be to perform analysis on the system you are trying to develop and then to design it.

Remember that in OO applications, the **most basic building-block in the system is the class**. Therefore, much of the effort in the OOAD process will involve identifying the classes required by the system and then deciding what they will contain in terms of attributes and methods (and possibly inner classes as we seen for GUIs).

A class and its contents can be described graphically in UML notation. For example, the UML **class diagram** of our Clock class is as follows:



This diagram can be broken down as follows:

● The top box in the class diagram contains the **name** of the class.

● The middle box contains the **attributes** of the class and their types. The **minus sign** in front of the attributes indicate that they are **private**.

● The bottom box contains the **method names**, **argument types and return type** (after the **colon**). The **plus sign** in front of the methods indicate that they are all **public**.

And that is pretty much all there is to documenting a class diagram using UML!

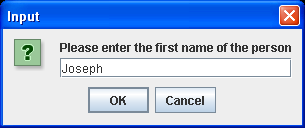
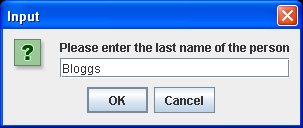
The big advantage of the class diagram is that, like pseudocode, it is possible to inspect it without having to worry about programming language syntax and specifics. Also, because the notation is **international**, it applies world-wide and so is completely **portable**. Indeed, tools exist that even allow you to take a class diagram as indicated above and have it translated directly into the OO language desired e.g. Java, C++ etc. in order to create “**skeleton classes**” – the programmer then fills in the missing bits, the method bodies (but also the logic for using the class and integrating it within the system as a whole, so there is still plenty to be done!)

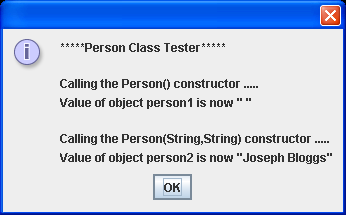
**Exercise 3**

Examine the UML class diagram below and use it to code the **Person** class it represents. Your Person class **should re-use existing methods in the class** as much as possible rather than referring to the class’s attributes directly. You should then write a driver class called **TestProgPerson** that fully tests your Person class. Write the driver as a **GUI** and get the user to supply a name when testing out the **Person(String, String)** constructor. Note that the no-argument constructor **Person()** should simply set the attributes to **empty strings**.

|  |
| --- |
| Person |
| - firstName : String  - lastName : String |
| + Person()  + Person(String, String)  + toString() : String  + setName(String,String) : void  + setFirstName(String) : void  + setLastName(String) : void  + getFirstName() : String  + getLastName() : String |

Your program should run as indicated in the following sample screenshots:



**Exercise 4**

Save the file **Person.java** as **Person2.java** and save the file **TestProgPerson.java** as **TestProgPerson2.java**.

Now redefine the class Person so that, in addition to what it already does, it also allows you to:

● store the middle name of a person i.e. an extra attribute

● retrieve and set the middle name of a person

● determine whether a given first name is the same as the first name of this person – you can call this method **matchesFirstName**()

● determine whether a given middle name is the same as the middle name of this person – you can call this method **matchesMiddleName**()

● determine whether a given last name is the same as the last name of this person – you can call this method **matchesLastName**()

● add a method **equals**() that returns true only if two Person objects contain the same first name, middle name and last name.

● add a method **makeCopy**() which takes a Person object as an argument and copies the attributes of the argument into this object.

● add a **copy constructor**, which takes a Person object as an argument and copies the attributes of the argument into this object.

**Note**: although the definition of the makeCopy() and copy constructor are the same, in order to test the copy constructor you will write code similar to the following:

Person person1 = new Person(“Joe”, “Anthony”, “Bloggs”);

Person person2 = new Person(person1);

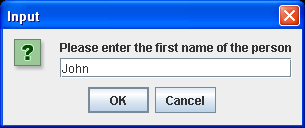
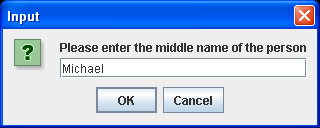
This creates a new Person object, referenced by person2 that will have exactly the same attribute values as those contained in person1.

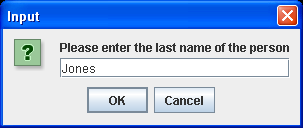
Your modified Person class should be taking maximum advantage of **code re-use**.

Of course, you now need to modify the corresponding driver class for the updated Person class. Ensure that it tests out the class fully.

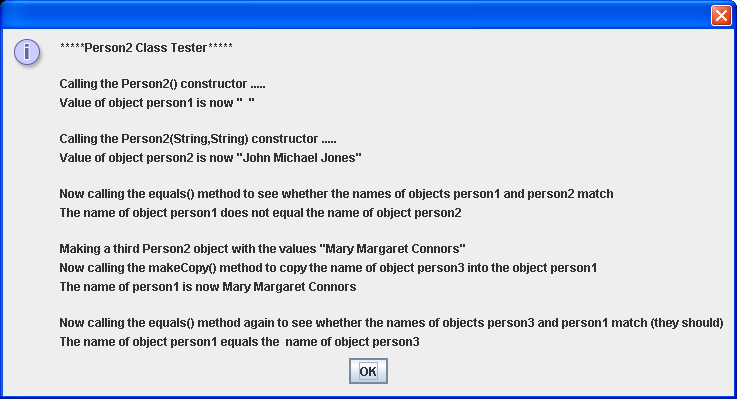
Your driver should be GUI based and run as indicated in the following sample screenshots:

**Entering the names for person2 object**



**Displaying the final output**



Note that the output screenshot makes no reference to the methods matchesFirstName(), matchesMiddleName(), matchesLastName() or a few other methods. This is because, in my case, they are being called (re-used) by other methods so, by testing the calling methods, you are automatically testing these methods. It is for the same reason that the mutators and accessors are not being tested in isolation – they are being called indirectly through the constructors and the toString() respectively.