**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming 1**

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

**Practical 4 – Introduction to Methods**

Over the next few lab sheets you will examine one of the most important program structuring units of all – the **method**.

**Writing Programs in Java Without Methods**

It is worth remembering that all programs in Java can be written using just the following:

• **sequence statements**

• **selection statements**

• **repetition statements**

**Sequence statements** are ones which just follow each other in a sequential fashion e.g.

System.out.print(“Please enter your age: ”);

age = input.nextInt();

**Selection statements** are ones which involve some sort of decision-making process using either some kind of if statement or a switch statement e.g.

if(age >=13 && age <= 19)

System.out.println(“A teenager”);

else

System.out.println(“Not a teenager”);

allows the program to decide whether or not a person is a teenager or not.

**Repetition statements** are ones which allow some iterative or repetitive process to occur in a program using either a while, do-while or for loop e.g.

letterAsString = input.nextLine();

character = letterAsString.charAt(0);

while(character < ‘a’ || character > ‘z’)

{

System.out.print(“Sorry - you must enter a lowercase letter! Please re-enter: ”);

letterAsString = input.nextLine();

character = letterAsString.charAt(0);

}

Any program can be written using appropriate combinations of these structures and they can be **nested** as deeply as required so we could have a while inside a for inside a do-while inside an if inside a switch etc (although things rarely get that complicated).

In fact in Java, **every selection can be made using only simple if statements**. Any if-else, nested else-if or switch can be written using only simple if statements, although not as nicely (or efficiently) in most cases.

Likewise, **every repetition can be made using only the while loop**. Any do-while or for loop can be written using an equivalent while loop instead, although not as efficiently in some cases, as we have seen (especially with the for loop).

**Why Methods?**

Most programs that solve real-life problems are much larger than the ones we have been analyzing so far. Experience in the software development industry has shown that the best way to develop, debug and maintain large programs is to construct them from small, simple pieces called **modules** – a programming process known as **modularisation**. Each **module contains a small, self-contained piece of functionality** that allows some specific task to be performed. In Java these modules are called **methods (**which belong to **classes)**. Modularisation also **promotes software re-use** since once a module/method has been written, it can be re-used as a building-block to create new programs.

**The Java API Pre-defined Methods**

From the very start of semester 1, every program we have written has used at least one method from the so-called **Java API** (Application Programmers Interface). The methods contained in the Java API are called **pre-defined methods** because, as programmers, **we do not need to write these methods ourselves**, we simply **borrow them into our program and re-use them** as we need them.

Some examples of pre-defined Java API methods that we have been using in the module are: print(), println(), nextInt(), nextLine(), format(), showInputDialog(), showMessageDialog(), charAt() etc. etc.

As you can see, we would never have gotten anywhere without the availability of these pre-defined Java API methods since most of our programs have been using some of them! **If such methods were not available, we would need to do the equivalent coding for them ourselves** – and that would not be very easy since many of the methods just mentioned are quite complex indeed – not something a novice programmer would easily tackle.

Therefore **pre-defined methods are vital** to the programmer and every programming language contains these “**libraries**” of methods. They are all about helping the application programmer and they **promote software re-use**. Next year, you will see that there are many other features within the Java language that promote software re-use but methods and classes are the main two.

Some **advantages** of the Java API pre-defined methods include:

● They have already been written by another programmer and so save us lots of coding effort and time when writing our own applications.

● Because they have been written by professional programmers, we can be confident that they have been tested fully and will operate correctly.

● Because they have been written by professional programmers, we can be confident that they have been well designed and are probably the most efficient way of accomplishing the task concerned.

● Because the Java API pre-defined methods are part of the Java language itself, using them means that are programs are **more portable**.

**Third-party Pre-defined Methods**

Apart from the Java API pre-defined methods, there are also thousands of methods available from other, third-party, sources such as books, CDs, web-sites, friends, companies that develop class libraries for certain applications etc. Again, these are pre-defined because we are not writing them ourselves from scratch. However, using third-party methods **may not always be as reliable** as those from the Java API (some may not work properly, some may not be very efficient etc.) and using them **will affect the portability** of your application because you are using methods outside of the Java system itself. We have not used any third-party pre-defined methods in this module so far and are unlikely to, but they can be very useful for certain applications.

**Organisation of Methods within the Java API**

Many programmers refer to the Java API as the **Java class library** because **every method defined in Java must belong to a specific class**.

We have been referring to this indirectly all along e.g. the print() and println() methods all belong to the **PrintStream** class and the format() method belongs to the **String** class (recall String.format()).

It is **crucial to know the class that a method belongs to**, so that you can use the method correctly – otherwise **syntax errors** are bound to occur.

There are literally **thousands of pre-defined methods within the Java API**. Ones that are somehow related are usually **grouped together within a certain class** and related **classes in turn belong to a specific package**. There are over a hundred different packages within the Java API.

**Package X**

**Class 1**

**Class n**

**Method 1**

**Method n**

**Method 1**

**Method n**

So, in reality, in order **to use a specific predefined method**, we must **import the package it belongs to** and we must **know the class it belongs to**.

Recall that we always import the javax.swing package whenever we wish to use the JOptionPane class. We wouldn’t be able to use the showInputDialog() or the showMessageDialog() methods without doing this at the top of the program.

As another example, methods that involve manipulating strings (such as charAt() and format()) all belong to the **String** class. The String class itself, because it is a general purpose class, belongs to the package called **java.lang**.

Interestingly, this package (because most programs require it) **does not need to be explicitly imported** into our programs – it is done automatically. **Every other package must be imported** however.

**User-defined Methods**

**User-defined** methods are ones which are **written by the application programmer** themselves (you!). It is usually necessary to write user-defined methods, especially where large programs are concerned, since they can help to make the program **more efficient** and the main() method **more readable**.

As mentioned already, for large software systems the use of methods becomes essential since they **allows a large programming task to be split up into several smaller, more manageable, programming tasks**.

Hence they provide an ideal means of **dividing a major software development job** **between several programmers**, each with the task of completing a certain part of the overall system. The component parts can be brought together eventually to form the completed system but each will normally be **tested in isolation** initially.

Of course, user-defined methods are also required because **there isn’t always going to be a pre-defined method already available that accomplishes the task we wish to perform**. For example, we can find the square-root and the cube-root of a number using the Math class predefined methods sqrt() and cbrt() but there is no provision for finding the fourth or fifth (or higher) roots in the same way. If we want to do this, we would need to write our own method that does it (although there is an indirect way to do it using another Math method)

You may not have realised it, but you have created your own user-defined method many times already. Each time you have written a program it has involved the use of the **main()** method. You have decided the contents of this method each time you wrote a program so you have defined what it contains. That said, the **main()** method is a **special** one in its own right since it has to be part of every Java application. Other than main() you have not created any other user-defined methods of your own, but that will change shortly.

The problem with writing user-defined methods is that they **must be tested** to ensure they work properly - just like all your programs need to be tested. Otherwise unexpected things could happen at run-time.

**Recognising and Calling Methods**

It is normally very easy to recognise a method in a line of code. The **parentheses** normally give it away e.g.

System.out.println(“Hello there everyone!”);

In this case the method println() is being **called**. We know that calling this method causes information to be displayed to the console window.

In general, when a method needs to be called, we supply the name of the method, then a left parenthesis, then zero or more **arguments**, then a closing right parenthesis.

**Method Arguments**

A method **argument** is a **piece of information** that a method needs in order to complete its task.

It may be useful to think about arguments in terms of inputs to a circuit. Here, the circuit has n inputs, labeled arg1, arg2, arg3 …. argn and produces at most 1 output.

arg1 arg2 arg3 arg4 …… argn

*method ()*

output

Think about the Math class’s square root method **sqrt**() for a minute. This method cannot complete its task unless you supply it with some number i.e. the square root method must be given an argument. Is one argument enough for sqrt()? The answer is yes because when we get the square root of a number, we mean “get the square root of one particular number”. If we do want to determine the square root of some other numbers, that’s no problem – we just re-use the sqrt() method again.

Now think about the method pow() from the Math class. This method should calculate the value of one number raised to the power of another number i.e. xy. Is one argument enough for this method to do its job? The answer now is no because it requires two arguments – the first might be the number itself and the second would be the power you are trying to raise the number to.

Calling the methods above would look as follows:

System.out.println(“The square root of 36 is ” + Math.sqrt(36));

System.out.println(“The value of 2 to the power of 5 is ” + Math.pow(2,5));

So, when a method needs to be called in a program, we must always call it with the **correct number of arguments**. If there is more than one argument to be supplied, then we must **separate the arguments up with commas** as in the case of pow() above.

It is vital also that the **type of value you give as an argument matches the type of argument expected by the method**.

For instance, both of the methods above expect to be given a **double** argument. Therefore, supplying values such as

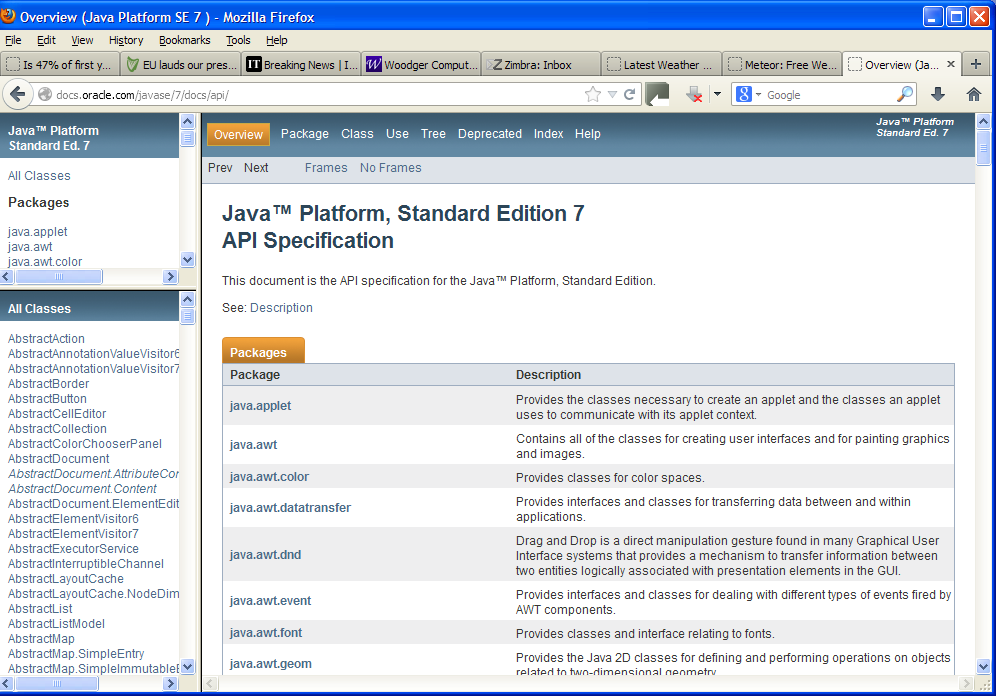
23.454, -234234.455, 65, 786.45f are all fine here because they are all valid double values (even though 65 is an int and 786.45f is a float, they are **promoted** to become a double when they are passed in as arguments here).

However, any non-number such as “one”, “22/09/2013”, “&\*” etc. would be rejected by the program and a **syntax-error** would occur indicating a **type mismatch**.

**Looking at the Java API Documentation**

In order to become proficient with the Java language it is essential that you are aware of the Java API and how to use it properly.

As mentioned already, the Java API comes with the Java language and there is also a comprehensive set of **documentation** available to programmers on every aspect of the API. This can be accessed from <http://java.sun.com/javase/7/docs/api/>



From the opening web-page above, you can see that the documentation is for the **Standard Edition 7** of the Java language – it is important that the documentation you are looking at actually matches the version of the Java language you are working with – otherwise discrepancies can occur.

You notice that there is an **alphabetical list of every Java class** that exists in the Java API on the left. In the main window there is an **alphabetical list of every package** that exists in the Java API, with a summary about what they are for.

Recall that we used the class **Font** recently, when we wanted to make sure that our tables got displayed nicely in a text area. This class belongs to the **java.awt** package. What do the classes contained in this package generally do?

**Organising your Work**

You should have a folder under X: called OOP1Stuff created. This time, create a folder called **Lab4** within OOP1Stuff to save your work from this lab session.

**Exercise 1**

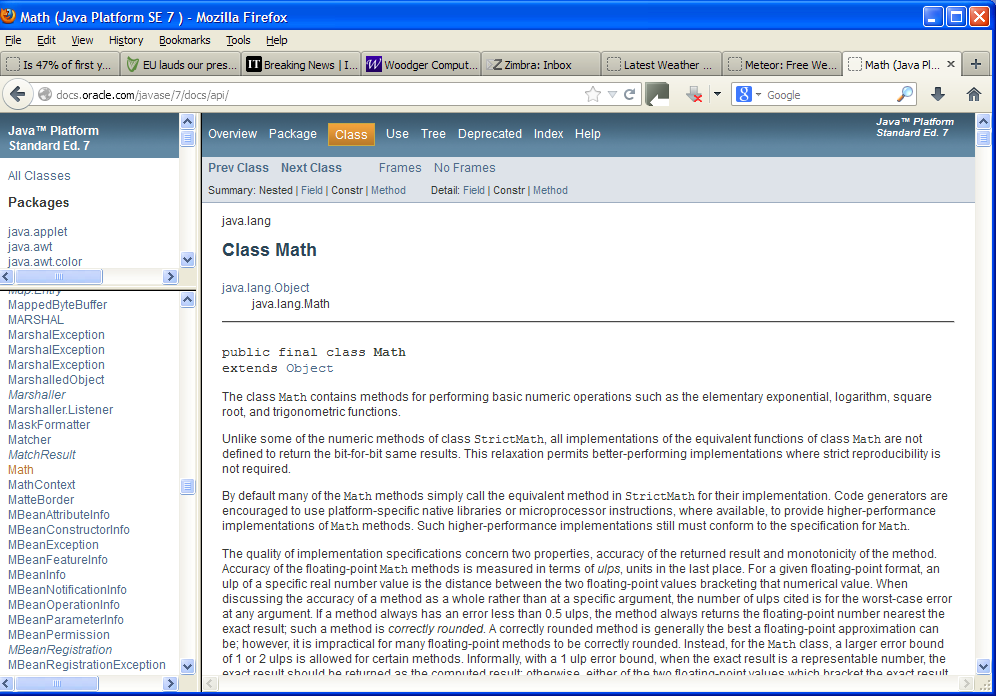
Scroll down the list here and find out what the classes in the following packages are for:

**java.net, java.sql, java.io**

**Checking out the Math Class**

Now you should **use the Java API documentation** to check out the **Math** class. Use the window on the left to find this and then click on the link.

You will see that there is a lot of information concerning the class.

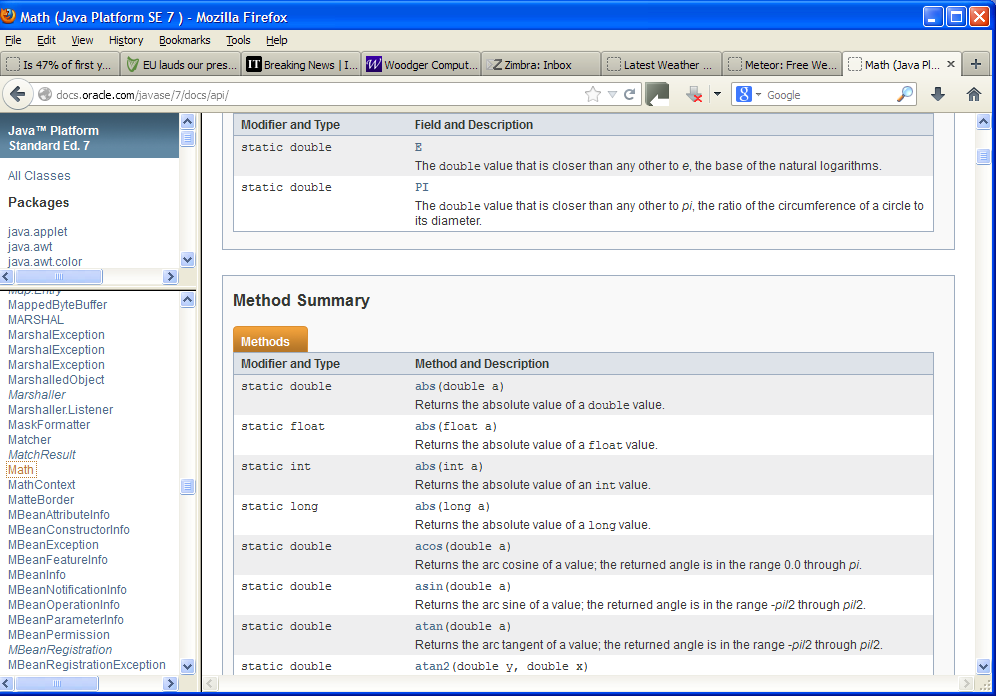


Notice that above the words “**Class Math**” are the words “**java.lang**” – this is the **package** to which the class actually belongs. As mentioned before, the Math class belongs to the java.lang package (the one we never need to import explicitly).

There is a great deal of information given here that is well beyond the scope of this module but still a good share of it relates to us. There is normally a description given about a class and its capabilities and this is the case here for Math.

If you scroll down a little more you will come to a “**Field Summary**” section. In the case of the Math class, this defines **numerical constants** that are defined within the Math class. These are PI and E. So you can see that Java has its own representation for PI, whereas up until now we have been creating our own constant for PI and setting it 3.142

After this is the “**Method Summary**” section. This is what we are really interested in at the moment. Have a look through the various methods available to us from the Math class. We can use every one of these methods should the need arise. We have already used some of them such as sqrt(), cbrt() and pow().



Look at the information given for the sqrt() method now – it says:

|  |  |
| --- | --- |
| static double | [**sqrt**](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#sqrt(double))(double a) |

What we are looking at here is called a **method definition header**. This gives the name of the method – **sqrt** – as well as the arguments expected by the method.

We can tell here by looking within the parentheses that the sqrt() method is expecting just the **one argument**. Also we can tell that the **type** of argument expected by sqrt() is a **double**.

Finally, we can tell that when the sqrt() method has completed its task, it will **return** a value of type double. The word “double” before the method name tells us this.

Because the sqrt() method is defined as a **static** method, we must **always put its class name before it** whenever we want to call it in a program i.e. we must have

**Math.**sqrt(100) rather than just sqrt(100)

In general, if a method is static and **belongs to another class** we refer to it as follows:

*Classname*.*methodName*(*arguments*)

Otherwise a **syntax error** will arise.

**Returning Values from Methods**

When a method is called it performs some task and, very often, returns a result to the point at which it was called. You can see that practically all the methods defined within the Math class **return a** **double** value.

So, when a line of code such as

System.out.println(“The square root of 36 is ” + Math.sqrt(36));

executes, what *really* happens is that, first of all, a **call** is made to the sqrt() method. It is **passed the value 36** as an argument. Because this is a valid number, it then executes the code within its method body which does the number-crunching on 36 in order to find its square root. Once the square root has been determined, the method then returns that value at the point it was called. So, at this point, the line of code above has really become:

System.out.println(“The square root of 36 is ” + 6);

What happens next is that the 6 is made into a “6” (this happens automatically when + is used) and finally this is joined onto the end of the text “The square ….” using the + operator. Finally, the println() method displays all of the characters to the console.

Another example is as follows:

sine = Math.sin(radians);

In this case the sin() method is called and it is passed as a value, the number stored in the variable radians (which is of type double). The sin() method then executes the code within its method body which does the number-crunching on the radians value in order to find its sine. Once this has been determined, the method then returns that value at the point it was called. So, at this point the line of code above has really become:

sine = *some number between -1 and +1*

What happens next is simply that the variable sine is assigned the value returned by the method sin().

**Methods that don’t Return Anything**

It should be noted that **not all methods return a value**. For an example, use the Java documentation now to look up the PrintStream class’s **println**() method. This is the method we have used often to display text to the console. The version of the method we have used the most has the following header:

|  |  |
| --- | --- |
| void | [**println**](http://java.sun.com/javase/6/docs/api/java/io/PrintStream.html#println(java.lang.String))([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html) x) |

Note that this method takes a single argument of type String and **returns nothing**. We can tell this because the **return type** before the name of the method is **void**.

A void method is one that returns nothing.

**Exercise 2**

There are a few methods called min() within the Math class. One of them has the following method definition header. Examine this now and answer the following questions:

|  |  |
| --- | --- |
| static float | [**min**](http://java.sun.com/javase/6/docs/api/java/lang/Math.html#min(float, float))(float a, float b) |

1. How many arguments does this method take?
2. What are the types of argument expected by this method?
3. What is the purpose of this method?
4. What is the type of value returned by this method?
5. Give a line of code that demonstrates the correct use of this method.

**Continuing the Examination of the Math Class**

The remainder of the Math class documentation contains the “**Field Detail**” and “**Method Detail**” sections. This is simply more detailed information about the fields and methods contained within the Math class. Normally we can get all the information we need from the “Summary” sections though. Note that what the documentation calls “**parameters**”, we are referring to as **arguments**.

**Exercise 3**

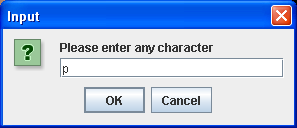
In the past, we examined a program to determine whether the character entered by the user was a digit. At the time, we used the code:

if (character >= ‘0’ && character <= ‘9’)

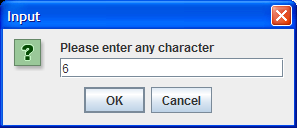
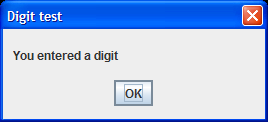
to verify the character was a digit.

However, there is already a Java API pre-defined method that exists in the **Character** class which allows us to do exactly the same thing. Use the Java documentation now to locate this method and, once you have figured out how to use it, utilise it in a program called **Exercise3.java** to determine whether or not the character entered was a digit. The program should run as indicated in the following sample screenshots – note that your program **does not require a loop**.

Run 1:

Run 2:

**Exercise 4**

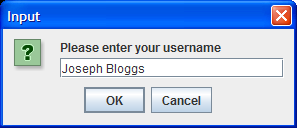
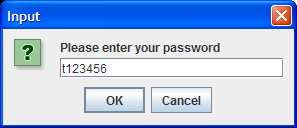
Imagine that a system is guarded by a username and password combination. Here the **username is case-insensitive but the password is case-sensitive**.

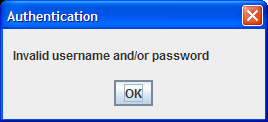
The user is prompted for each and these are then compared to values within a database containing valid username and password combinations. For the purposes of this program you can take it that the valid username is “Joe Bloggs” and the valid password is “t123456”.

The **user should only get one attempt** to enter the correct combination here (so no loop is required).

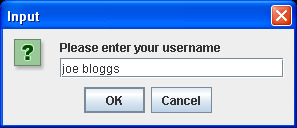
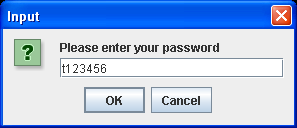
There is a method within the **String** class that can be used to determine whether the username is valid (ignoring case). Use the Java API documentation to find this method and use it in your program to validate the username. Your program should be called **Exercise4.java** and run as indicated in the following sample screenshots:

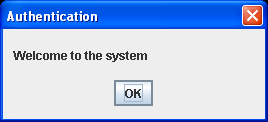
Run 1:



Run 2:

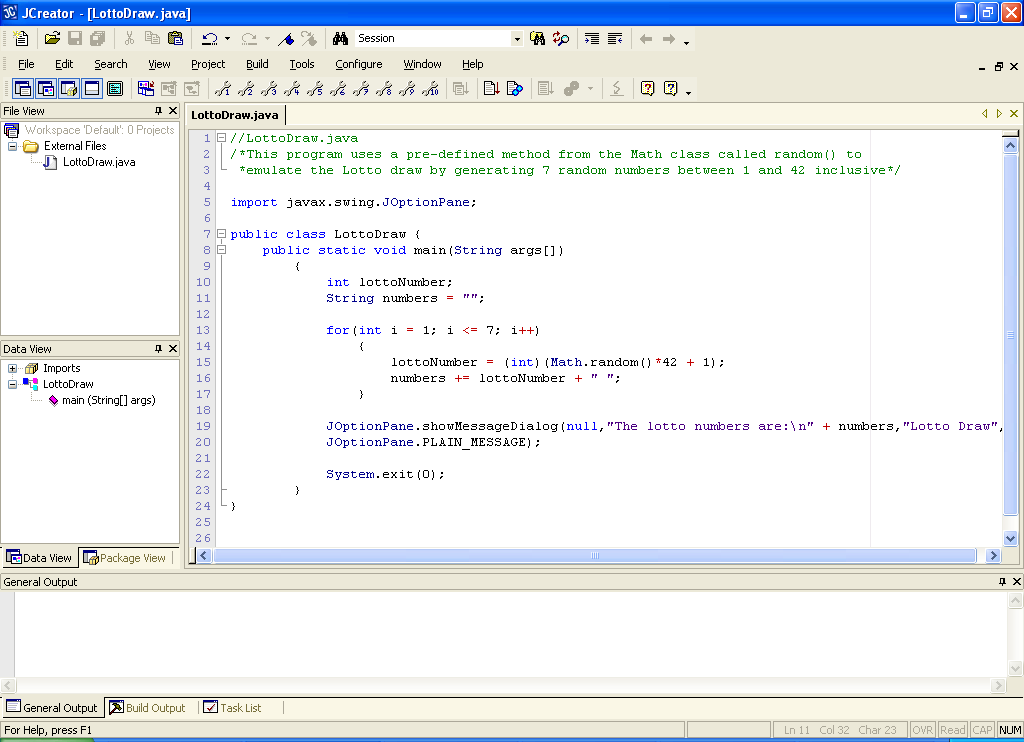


**Random-number Generation**

The ability to generate random numbers has many applications in software development, in particular with games and simulation type programs.

**Aim:** We wish to write a program using a for loop which simulates the lotto draw by selecting 7 random numbers from 1-42.

**Java Code**:



**Analysis of program:**

• Here the loop iterates a total of 7 times. Each time the loop iterates a random number between 1 and 42 inclusive is generated using the code:

**(int)(Math.random()\*42 + 1)**

This code might appear a little strange at first glance but it is quite straightforward once broken down.

It begins with the **random()** method generating a random number between 0 and, up to **but not including** 1 i.e. 0.9999999.

This number is then multiplied by 42 to generate a random number between 0 and 41.9999999.

This number is then truncated i.e. the fractional part is “chopped off” via the **(int) type-casting** to give us a random integer number between 0 and 41 and finally 1 is added to this to give us a random integer number between 1 and 42 inclusive.

• As soon as the random number is generated, it is joined onto the String variable numbers with the code:

**numbers += lottoNumber + " ";**

Notice how the **arithmetic assignment operator** **+=** can be used with String variables.

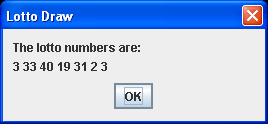
• Once the loop stops the 7 randomly generated numbers are displayed on a message dialog.

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

Click the **New File** icon on the JCreator IDE and save the file as **LottoDraw.java** in your Lab4 folder. Now type in the code for the program above.

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 several times to test it fully. You should see that on certain runs of the program some of the numbers appear twice or even three times as indicated below!



**Improving the Last Program**

The above solution is not perfect because there is no guarantee that 7 different numbers will be generated when the program runs. To solve this, one possibility is to declare 6 integer variables and set them all to zero (or any other value that could not be one of the lotto numbers) to begin with. These same variables will be used to store the numbers generated as we go along and, for the next number generated, check that it is not equal to any one of the previously generated numbers.

If it is, then loop continuously, generating a new random number each time until a number is generated that is different from the previous ones “picked”. Think about it for a while – it is certainly tricky.

You should now save the last program as **LottoDrawImproved.java** and make the necessary modifications. Run the program about 10 times to make certain that no set of 6 numbers contains duplicates.