**Institute of Technology Tralee**

**Computing Department**

**Object Oriented Programming 1**

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

**Practical 8 – Input Validation**

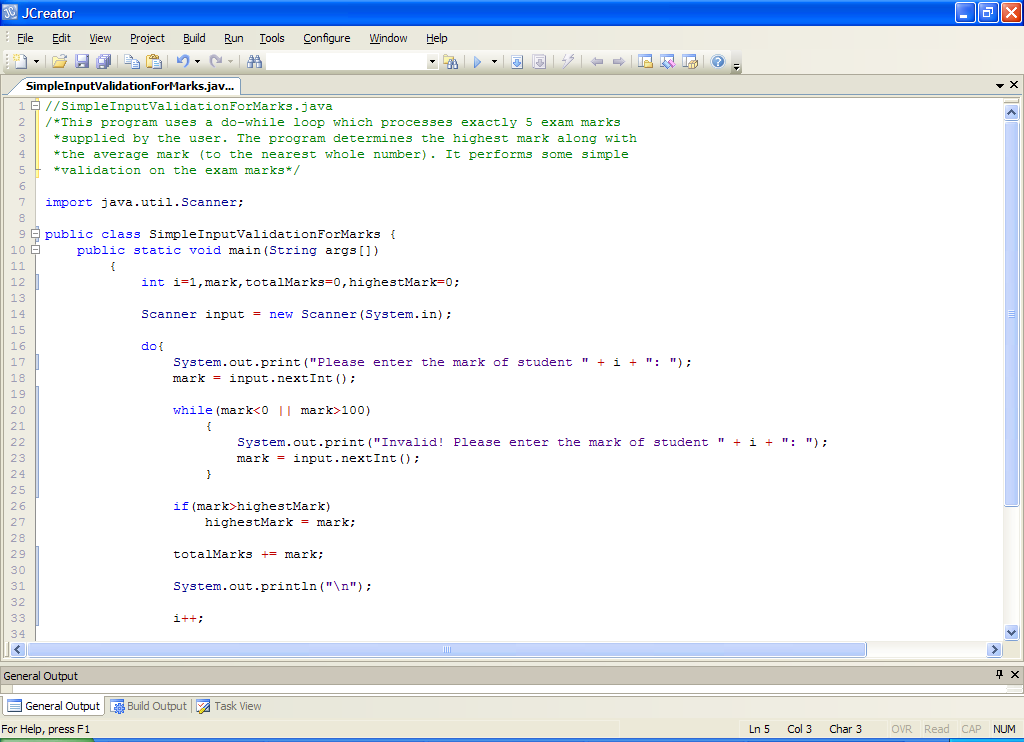
With the first assessment completed, we now turn our attention to another vital part of software development – making sure that our programs are as foolproof as possible. We do this with proper **input validation algorithms** in our code. We won’t cover any new structures in this section but we will be putting all our prior knowledge and **problem-solving skills** to use in order to make our programs as robust as possible.

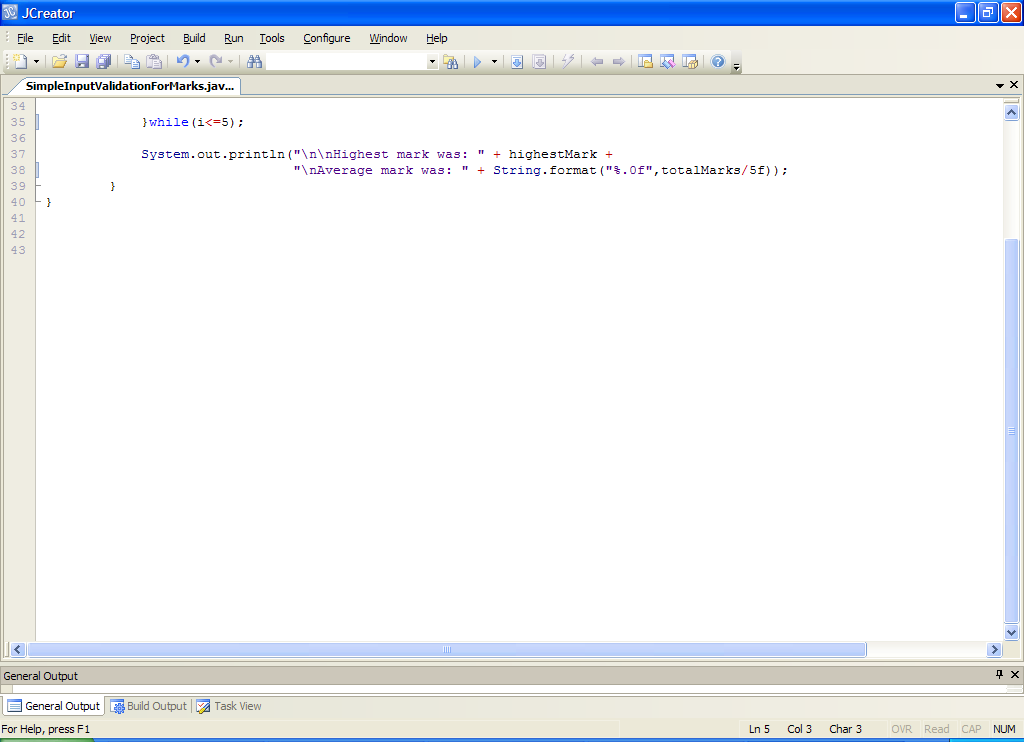
**Input Validation**

You have written many programs up to this time which have been quite vulnerable in the sense that some incorrect inputs could have caused them to ‘crash’ and still others would have produced **logical errors**. We will examine some of these over the next few lab sheets also. You have all seen this at one time or another through testing your programs. However, if we can apply **input validation** to our programs, we are in a very good position since we can actually prevent invalid values from being accepted by the program in the first place and hopefully avoid the possibility of a system crash and various logical errors, which is essential where industrial-strength applications are concerned.

**Aim:** We wish to read in the exam marks for 5 students and determine the highest mark achieved along with the average of the marks to the nearest whole number. These marks should all be mathematical **integers** (**whole numbers**) within the range 0 to 100. All values outside of this range should be rejected and the user asked for a different value instead. This first attempt performs simple validation, where we trust the user to at least supply a whole number to the program at runtime for each mark.

**Java Code:**





**Analysis of program:**

• we create 2 counter variables here. One is the loop variable, **i** and the other is for keeping track of the total of the marks as they are entered (so that we can calculate the average of the marks later). We also have a variable which stores the highest mark encountered and initialize this to zero.

• For variety, we use a counter-controlled **do-while** loop for the main iteration process here. This loop will iterate 5 times in total. It begins by asking the user to enter an exam mark. We trust that the user will enter an integer value here when asked for the mark.

If they do, then we perform some simple input validation on the value supplied with the code:

**while(mark<0 || mark>100)**

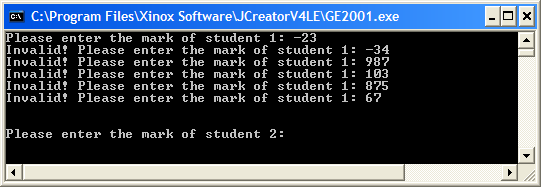
**{**

**System.out.print("Invalid! Please enter the mark of student " + i + ": ");**

**mark = input.nextInt();**

**}**

This is a data-sentinel controlled while loop that keeps going as long as the value entered by the user lies outside the range 0-100 i.e. as long as the value entered remains invalid. The message inside the loop tells the user that the mark is invalid and asks them to re-enter. Eventually, when the user does enter a valid value, the loop stops and the rest of the mark’s processing occurs. The screenshot below shows an example where several invalid marks have been entered in sequence and eventually a valid mark of 67 is entered.



• The next bit of processing on the mark is to try to keep track of the highest mark entered. This is achieved with the simple if:

**if(mark>highestMark)**

**highestMark = mark;**

A test is performed to see whether the mark just entered is greater than the current highest mark. If it is, then we reset the variable highestMark to this newly entered mark.

• The next code in the main loop is

**totalMarks += mark;**

which keeps a running total of all the marks as they are entered. It uses the arithmetic assignment operator for addition **+=** as a shortcut.

• Once the **do-while** loop has finished, the program then displays the highest mark entered and the average of the marks to the nearest whole number using the **format**() method and the **%.0f** specifier.

**Organising your Work**

Within OOP1Stuff create a folder called **Lab8** to save your work from this lab session.

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

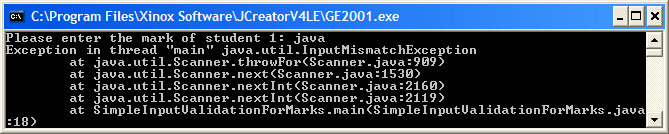
Click the **New File** icon on the JCreator IDE and save the file as **SimpleInputValidationForMarks.java** in your Lab8 folder. Now type in the code for the program above.

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 and test it fully. You will see that should the user enter any non-integers then things go a bit pear-shaped. We analyse this scenario in a little detail next so that we can learn a little more about **runtime errors**.

**Runtime Errors in the Previous Program**

If the user doesn’t enter a proper integer value, for example, should they enter a word such as “java”, then the following will happen at runtime:



The program has **crashed** here, and the Java runtime system displays what is known as a “stack trace” which gives a list of the methods that were called on the way to the crash actually occurring. Often, it is the first and last lines of the stack trace that really matter in terms of determining what went wrong and trying to fix the problem. So here, the first line of the stack trace says:

**Exception in thread “main” java.util.InputMismatchException**

This tells us the **type** **of runtime error** that occurred here – it was an “InputMismatchException”.

The last line of the stack trace says:

**at SimpleInputValidationForMarks.main(SimpleInputValidationForMarks.java:18)**

This tells us the line in our program where the error occurred. In this case it was on **line 18** of the program. If we now look to line 18 we see the code:

**mark = input.nextInt();**

there is nothing wrong with the code, and that is the reason it has compiled fine. But it causes a crash at runtime here because the type of **input** supplied by the user **does not** **match** the type of input expected by the **nextInt**() method, hence the InputMismatchException.

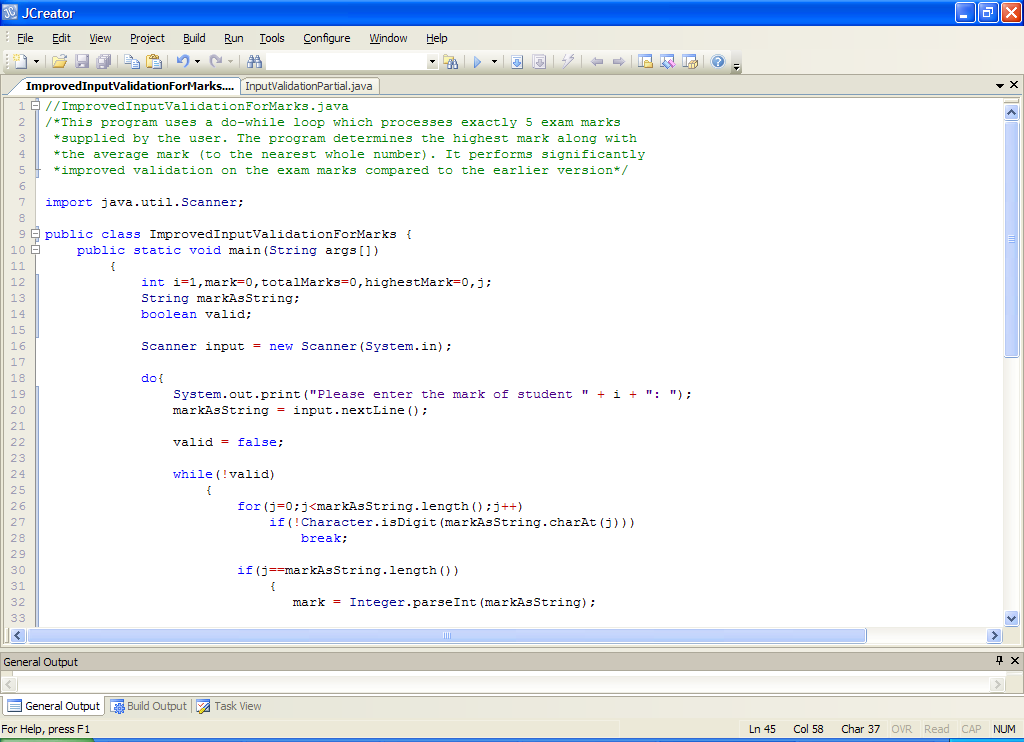
The only way we can prevent such an exception from occurring is by ensuring that the values entered by the user are valid integer values.

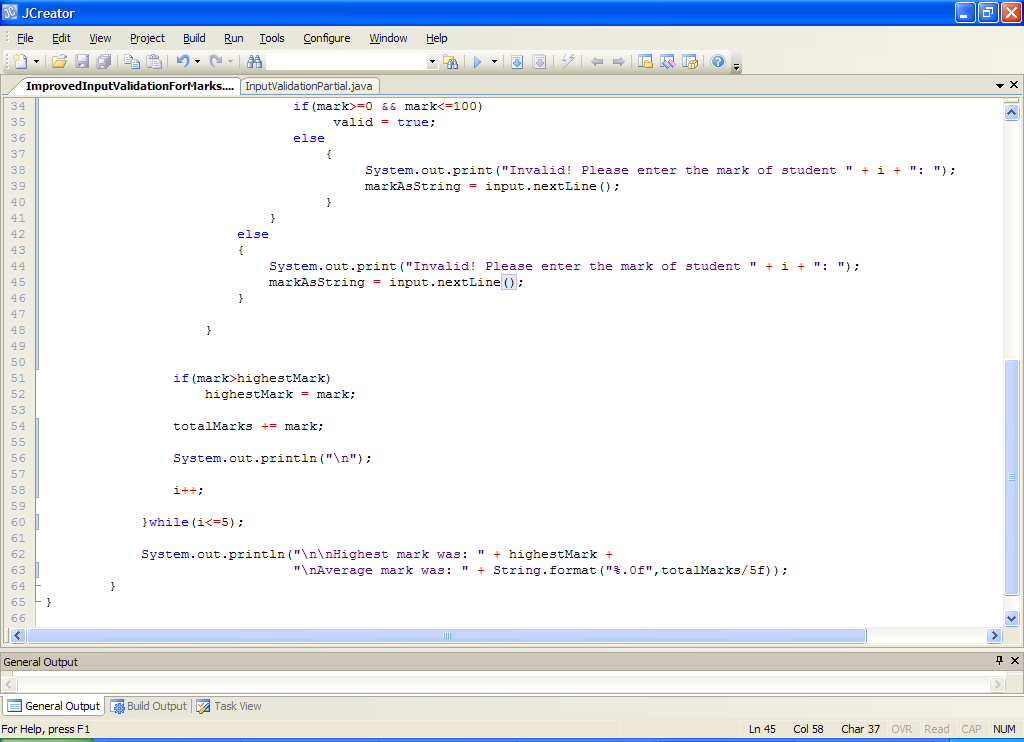
Our next attempt at input validation will aim to improve significantly on the current situation.

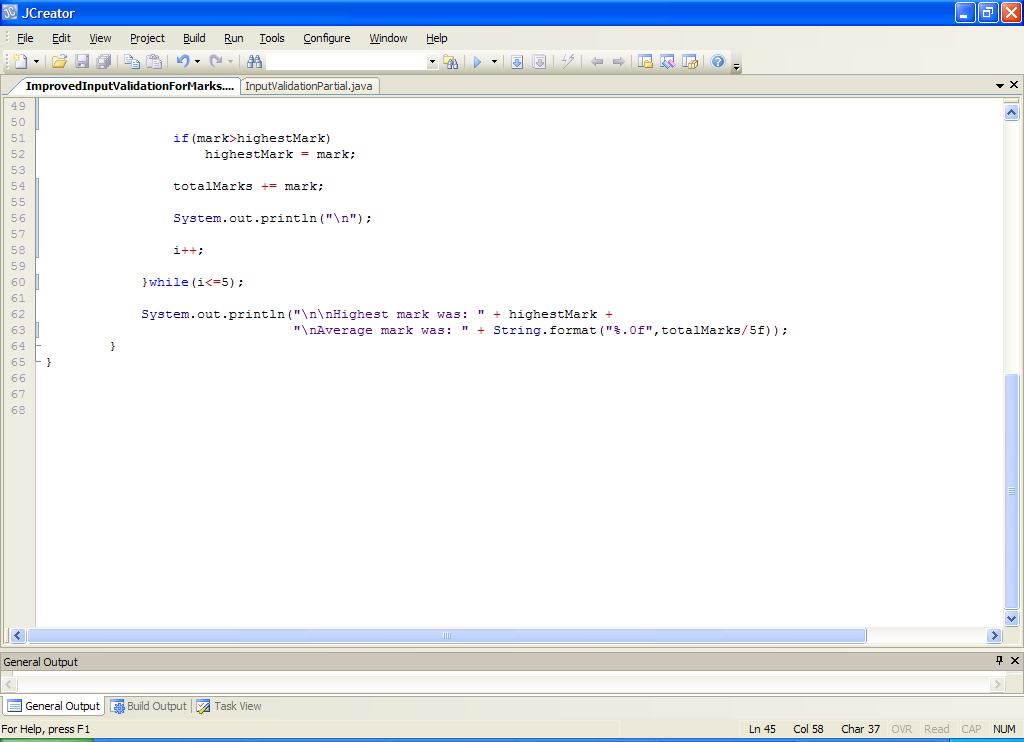
**Improving the Validation of Exam Marks**

**Aim:** We wish to read in the exam marks for 5 students and determine the highest mark achieved along with the average of the marks to the nearest whole number. These marks should all be mathematical **integers** (**whole numbers**) within the range 0 to 100. All values outside of this range should be rejected and the user asked for a different value instead. This second attempt performs significant validation, where we check each mark entered to make sure it is definitely a positive whole number and, if so, make sure it is also within the 0-100 range.

**Java Code:**







**Analysis of program:**

• We have an extra counter variable now, this is j. This will be used in the **for loop** later.

• There is also a **boolean** variable called **valid**. We have discussed this **primitive data type** before when talking about boolean methods but we are using in here in a different context.

Recall that boolean variables can have only one of two values – **true** or **false**. We use it in this program to **keep track of whether the number entered by the user is a valid mark or not** – we must assume, to begin with, that any particular input value is not a valid mark and so we set the value of valid to **false** just before the validation loop begins on line 24. After performing some tests, if we determine it is a valid mark, then valid is set to **true** and the inner validation while loop will stop, otherwise the user will get the “Invalid” message and the while loop keeps going. This validation loop is a **task-controlled while loop** where the loop keeps going until the task is completed, the task in this case getting completed when the value of the variable valid becomes true.

• At the start of the do-while loop, the user is prompted for a mark and this is read in and stored in **markAsString**. Recall in the last version of the program, we read the mark in using nextInt(). The difference is that in the original version, we assumed the user would enter a whole number, but now we must assume the worst and take it that the user could enter anything. As we saw, if we were to continue using nextInt() our program would crash on non-integer inputs and so we must use nextLine() here to read in the mark as a string in the first instance. If it should turn out that the mark is indeed a valid one, then we will convert the string version of it to an integer version, but this will only happen when our validation routine has inspected the mark.

• Once the mark is read in (as a string), the boolean variable is set to false, as mentioned before. This **ensures we will get into the validation loop** for each newly entered mark, and never exit it until a valid mark has been entered, at which point the variable valid is set to true and therefore the loop expression becomes **(!true)** which is just **(false)** and so the validation loop comes to a halt. Notice how the **logical not operator** is used to create the validation loop test expression **!valid** here. It could also have been coded as **(valid == false)** instead.

• At the very start of the while (validation) loop, there is yet another loop! This is a **for loop** that iterates a certain maximum number of times dictated by the number of characters contained in the users input value. This number of characters is determined by the **length**() method used here. As you know, this method is part of the String class and so can be called on String variables (object references) such as markAsString.

**j < markAsString.length()**

• After this the code

**if(!Character.isDigit(markAsString.charAt(j)))**

**break;**

checks to see whether the character at position j in the users input is a digit. If it is not, then the test expression here evaluates to true and we immediately stop testing subsequent characters with the **break** statement. This break is important from the point of view of **efficiency** because if we determine that a character is not a digit, then it cannot be a valid integer and so there is no point in wasting CPU time checking other characters.

If the character does turn out to be a digit, then the break statement will be bypassed and then the loop variable j will be incremented with the **j++** code.

• After this for loop has completed, we can then determine whether or not the user input a valid positive integer. If they did, then the loop must have completed its full cycle and the value of the **variable j must be equal to the value of markAsString.length()** i.e. the number of characters contained in the users input. In this case we now convert the mark from string to integer form using the parseInt() method. We can do this now as we can be confident that the program shouldn’t crash (technically it still could, but we will deal with those possibilities later). If the for loop exited early, then j will be less than the value of markAsString.length() and so the value entered could not have been valid, so the user gets an “Invalid!” message and is asked to re-enter the mark.

• If we are dealing with a valid positive integer, we just need to check that the integer lies in the range 0-100 with the code

**if(mark>=0 && mark<=100)**

**valid = true;**

so the variable valid gets set to true if it also passes this test. Otherwise the user will get an “Invalid!” message and be asked to re-enter.

• The remainder of the code is exactly the same as in the previous version.

**Take your time** trying to understand the validation routine here, as it is not trivial. Time spent figuring out the operation of this routine will help you as we investigate other problems.

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

Save the file **SimpleInputValidationForMarks.java** as **ImprovedInputValidationForMarks.java**  in your Lab8 folder and make the necessary modifications.

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 with the following set of test values:

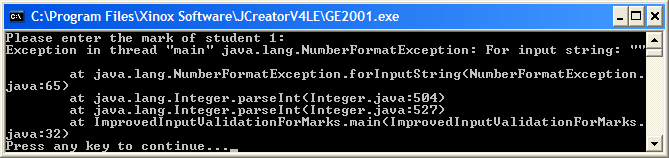
23, 45, 98, 123, 76.45, -889.3, thirteen, 34.5456f, abcde, %£”, -234, “” (i.e. hit return), 2500000000

**Remaining Flaws in the last Program & Run-time Errors**

Although the program is very good in terms of fool-proofing, it is not perfect, as you can see. For example, if the user just hits return on the keyboard, the program crashes. Likewise, if the user enters a value like 2500000000 the program also crashes. Thankfully, both of these issues are relatively easy to sort out – definitely a lot easier than the improved validation routine we have discussed already. However, sorting both requires some idea of what went wrong at runtime.

**Entering the Empty String**

In the case of hitting return on the keyboard, the following is produced at runtime:



Again, we focus our attention on the first and last lines of the “stack trace” produced. This tells us that a **NumberFormatException** occurred when the empty string “” was supplied and that the program **crashed at line 32** of our program. If we look at line 32 we see the following code:

**mark = Integer.parseInt(markAsString);**

so the program crashed when an attempt was made to convert the empty string to an integer. So we can prevent this runtime error if we can stop the program getting this far when we supply the empty string at runtime. It might seem a bit surprising that the program does get this far when empty string is supplied as input but remember that the empty string has a length of zero, so the code:

**for(j=0;j<markAsString.length();j++) if(!Character.isDigit(markAsString.charAt(j)))**

**break;**

will check to see if j (which is zero) is less than markAsString.length() – which is also zero in this case. This is false (because zero is not less than zero) and so the loop immediately exits. This isn’t really a problem until we come to the next section of code:

**if(j==markAsString.length())**

**{**

**mark = Integer.parseInt(markAsString);**

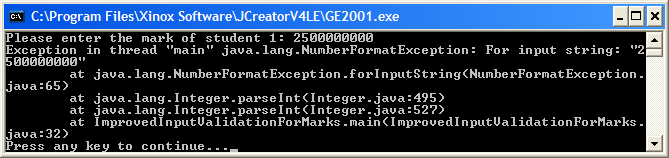
**//rest of if section**

now the value of j is still zero and markAsString.length() is also zero so they match which according to the logic means we are dealing with a valid integer (which of course we aren’t in this case). Therefore an attempt is made to convert the empty string to an integer and the call to parseInt() causes a NumberFormatException to occur.

So how do you think you can sort this issue out? Well, one quick way is to **modify the if test expression** above to take the possibility of the empty string into account.

**Entering a Very Large Positive Integer value**

The other remaining problem with the validation routine is that large integer numbers such as 2500000000 would cause the program to crash also. Again, we will examine the stack trace to see what happens at runtime:



Looking at the first line of the trace we see that a NumberFormatException has occurred for the input string supplied and the last line tells us that the runtime error occurred on line 32 of the program. So the stack trace here is identical to the previous case of entering the empty string. Again, the issue is that our logic treats this number as a valid integer (which it actually is in this case) and as far as the code:

**mark = Integer.parseInt(markAsString);**

on line 32 where the crash occurs. So, even though 2500000000 is a valid mathematical integer value, Java finds a problem with it. The reason is relatively easy to understand here. In Java, like all programming languages, primitive data types such as int are **allocated only a certain amount of memory** in order to store their values. In Java, an int can store 32 bits of information. As there can be positive or negative integers, this means that in Java, the maximum valid int can only be 232/2 – 1 i.e. 2,147,483,647 and the minimum is -2,147,483,648. In other words valid integers in Java go from roughly -2Billion to +2Billion. The number we supplied here was 2.5 Billion so is too large to be accommodated and the program duly crashes. The Java **Integer** class defines the maximum and minimum value an int can store with the **constants**

**Integer.MAX\_VALUE** and **Integer.MIN\_VALUE**

You can use these shortly in your solution.

So how do we fix this problem? One way of doing it quickly involves taking advantage of another primitive data type, the **double**. In Java, a double can store 64 bits of information in such a way that it allows for a maximum value of around 1.79769x10308

and a minimum value of around -1.79769 x10308. Like the int equivalents, you never need to remember these numbers as they are defined as **constants** within the Double class as follows:

Double.MAX\_VALUE and -Double.MAX\_VALUE

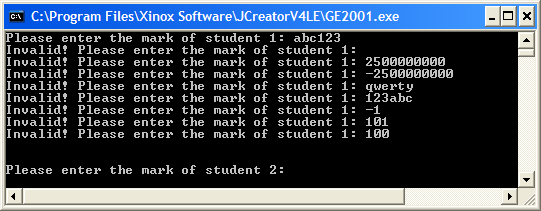
These numbers are absolutely massive and would cover practically any circumstances when programming.

All we need to do now with our validation routine is to check to see whether the user entered a valid positive mathematical integer as before and, if it is valid, **convert it from a String to a double**. Then, as long as the double value lies within the range **Integer.MIN\_VALUE** to **Integer.MAX\_VALUE**, we know we have a valid Java int on our hands so we can then safely convert the double to an int via **type-casting**.

Strictly speaking the conversion from the double to int is not essential in terms of validation, but we do it here as other variables in the program such as totalMarks and highestMark were already integers and we want to avoid making these double also.

**Fixing the Flaws with the Program**

Now take the current version of the program and save it as **PerfectInputValidationForMarks.java** – see if you can make the necessary modifications so that when you enter a valid positive mathematical integer (but invalid positive Java integer) the user gets the “Invalid!” message and also, if you enter the empty string, you get the “Invalid!” message. **Deal with the empty string problem first as it is much easier to solve**. The program will now run as follows:



**Re-using the Validation Routine Developed**

As you know well by now, there are almost always different solutions to the same problems. The validation routine we have examined in detail here is very useful as it can be used in any situation where we need to get the user to enter whole numbers of any size e.g. if you just wanted to write a program to read in an arbitrary amount of valid positive whole numbers and get their average, we could use the algorithm above as a basis. The only thing we would bypass is the check to see whether the number lies in the range 0-100 as this is specific to the validation of exam marks.

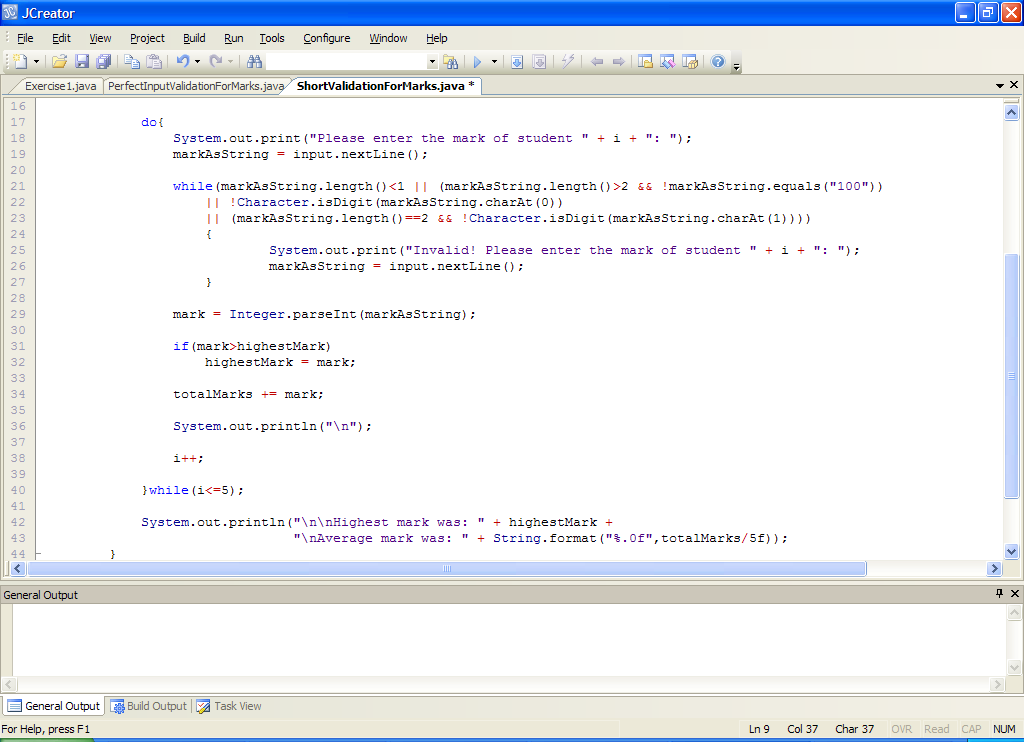
So, although the validation routine works perfectly, and can be easily modified for other situations, we might have been able to code a more straightforward solution in the case of exam marks, given the restrictions that apply.

**An Alternative Perfect Input Validation Routine For Marks**

In the case of exam marks, we note that valid inputs must all be in the range 0-100. This is a massive restriction on the allowed input values and one that we can take advantage of to code a much shorter validation routine. Basically we can say that:

* Valid marks must have a length between 1 and 3 characters inclusive
* If the length is 3, then the only valid mark is 100
* All characters in the mark must be digits

This leads us on to the alternative validation algorithm you see below:



So here we enter a mark and then a **data-sentinel controlled while loop** with a lot of sub-expressions in its test begins.

* The first sub-expression here checks to see if the **length of the mark entered is less than 1** – if it is then we’ve entered the **empty string** so the user would get the “Invalid!” message.
* The next sub-expression checks to see if the **length of the mark entered is greater than 2** and, if it is, it checks to see whether the **mark was not 100**. If the mark was not 100, then we’ve entered an invalid input so the user gets “Invalid!” This test rules out loads of bad inputs, including the **very large positive and negative integers** we had to deal with before.
* The next 2 sub-expressions are related and are checking to see whether **the characters making up the mark are all digits**. If any of them are not, then the “Invalid!” message gets issued again. Note that for the 2nd sub-expressions here, we need to make sure that we are dealing with a mark of length 2 *before* we try to extract the second character with **charAt(1)**. Otherwise, the **program will crash** if we attempt to extract a character from a position that does not exist within the string.

You can see that the solution here saves us the trouble of the boolean variable, some other variables, the for loop, the test after the for loop and the tests against Integer.MAX\_VALUE and Integer.MIN\_VALUE.

It is very important that you try to keep your mind open when solving problems of this nature. Think long and hard about how you might solve a given problem and also think about whether there might be alternatives to your first idea. There usually is. From the point of view of the next assessment, **the crucial thing is that the validation routine works correctly**.

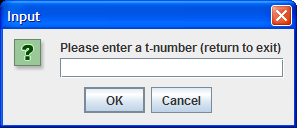
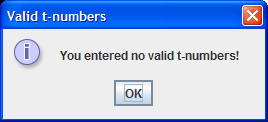
As a thought exercise now just for a few minutes (and for later on), see if you can come up with any other possible validation routine that will work for marks. I can think of one right now that is **really simple** in terms of the algorithm involved, and it will work, but it **isn’t very efficient** as it involves loads of tests! Can you guess what this routine looks like?

You should now take the **PerfectInputValidationForMarks.java** program and save it as **ShortValidationForMarks.java** – make the necessary modifications as outlined in the validation routine above. You can also remove any unnecessary variables that were in the original code. Test the program with a broad range of input values to mark sure it is passing each one.

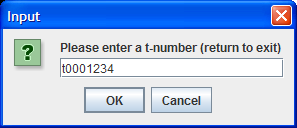
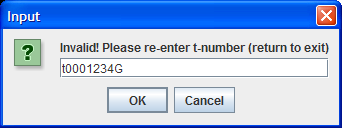
**Exercise 1**

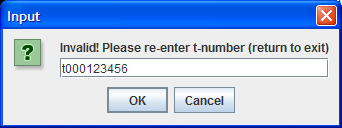
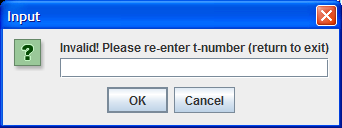
Every student at the ITT receives a unique t-number on registration. A valid t-number begins with the letter t (uppercase or lowercase) followed by exactly 8 digits. Write a program that prompts the user for an arbitrary amount of t-numbers (end of input is signaled by the user hitting return) and validates each one perfectly. Once the main loop has completed, the program should display the total number of valid t-numbers entered along with a list of them. If the user enters no valid t-numbers, the program should just issue a message indicating this. Spend some time thinking about your validation algorithm here – it doesn’t matter how you code it as long as it is logically correct. Your program will run as indicated below:

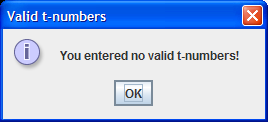
Run 1: The user hits return immediately

Run 2: The user enters 3 bad t-numbers, then hits return



Run 3: The user enters 3 valid t-numbers, then hits return

