JAVA with NETBEAN

One of the difficult things about getting started with Java is installing everything you need. Even before you write a single line of code, the headaches begin! Hopefully, the following sections will make life easier for you.

We're going to write all our code using a free piece of software called NetBeans. This is one of the most popular IDEs (Interface Development Environment) in the world for writing Java programmes. You'll see what it looks like shortly. But before NetBeans will work, it needs you to install the necessary Java components and files. First up is something called the Java Virtual Machine.

The Java Virtual Machine

Java is platform independent. This means that it will run on just about any operating system. So whether your computer runs Windows, Linux, Mac OS, it's all the same to Java! The reason it can run on any operating system is because of the Java Virtual Machine. The Virtual Machine is a programme that processes all your code correctly. So you need to install this programme (Virtual Machine) before you can run any Java code.

Java is owned by a company called Sun Microsystems, so you need to head over to Sun's website to get the Java Virtual Machine, also known as the Java Runtime Environment (JRE). Try this page first:

http://java.com/en/download/index.jsp

You can check to see if you already have the JRE on your computer by clicking the link "Do I have Java?". You'll find this link under the big Download button at the top of the page. (Unless Sun have changed things around, again!) When you click the link, your computer will be scanned for the JRE. You will then be told whether you have it or not. If not, you'll be given the opportunity to download and install it.

Or you could just head over to this page:

http://java.com/en/download/manual.jsp

The "manual" in the above links means "manual download". The page gives you download links and instructions for a wide variety of operating systems.

After downloading and installing, you may need to restart you computer. When you do, you will have the Java Virtual Machine.

The Java Software Development Kit

At this stage, you still can't write any programmes. The only thing you've done is to install software so that Java programmes can be run on your computer. To write code and test it out, you need something called a Software Development kit.

Java's Software Development Kit can currently be downloaded from here:

http://developers.sun.com/downloads/

The one we're going to be using is called Java SE. (The SE stands for Standard Edition.). Click on that link, then on "Java SE (JDK) 6 Download". You'll then find yourself on a page with a bewildering list of options to download. Because we're going to be using NetBeans, locate this:

JDK 6 Update X with NetBeans 6.x

Click the Download link to be taken to yet another page. Click the top download to be taken to a page that asks you to select your operating system. Click Continue to finally get the download you need. A word of warning, though - this download will be big, at over a 130 megabytes at the time of writing! Once you've downloaded the JDK and NetBeans, install it on your computer.

We're going to be using NetBeans to write our code. Before launching the software, however, here's how things work in the world of Java.

How things work in Java

You write the actual code for your programmes in a text editor. (In NetBeans, there's a special area for you to write code.) The code is called source code, and is saved with the file extension .java. A programme called Javac is then used to turn the source code into Java Byte Code. This is known as compiling. After Javac has finished compiling the Java Byte Code, it creates a new file with the extension .class. (At least, it does if no errors are detected.) Once the class file has been created, it can be run on the Java Virtual Machine. So:

- Create source code with the extension .java
- Use Javac to create (compile) a file ending in .class
- Run the compiled class

NetBeans handles all the creating and compiling for you. Behind the scenes, though, it takes your sources code and creates the java file. It will launch Javac and compile the class file. NetBeans can then run your programme inside its own software. This saves you the hassle of opening up a terminal window and typing long strings of commands,

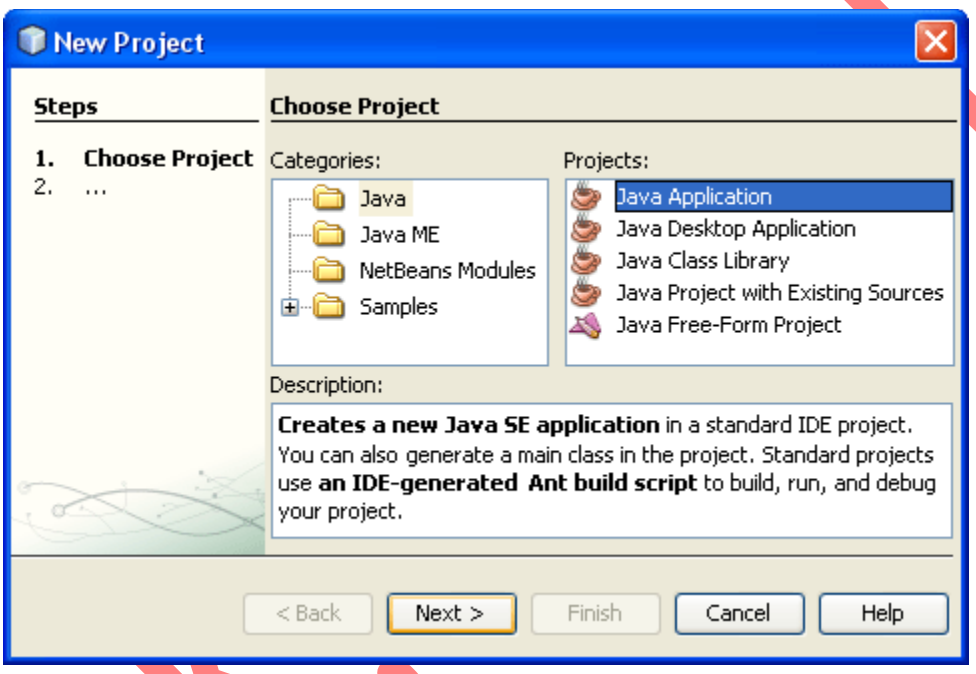
The NetBeans Software

When you first run NetBeans, you'll see a screen something like this one:

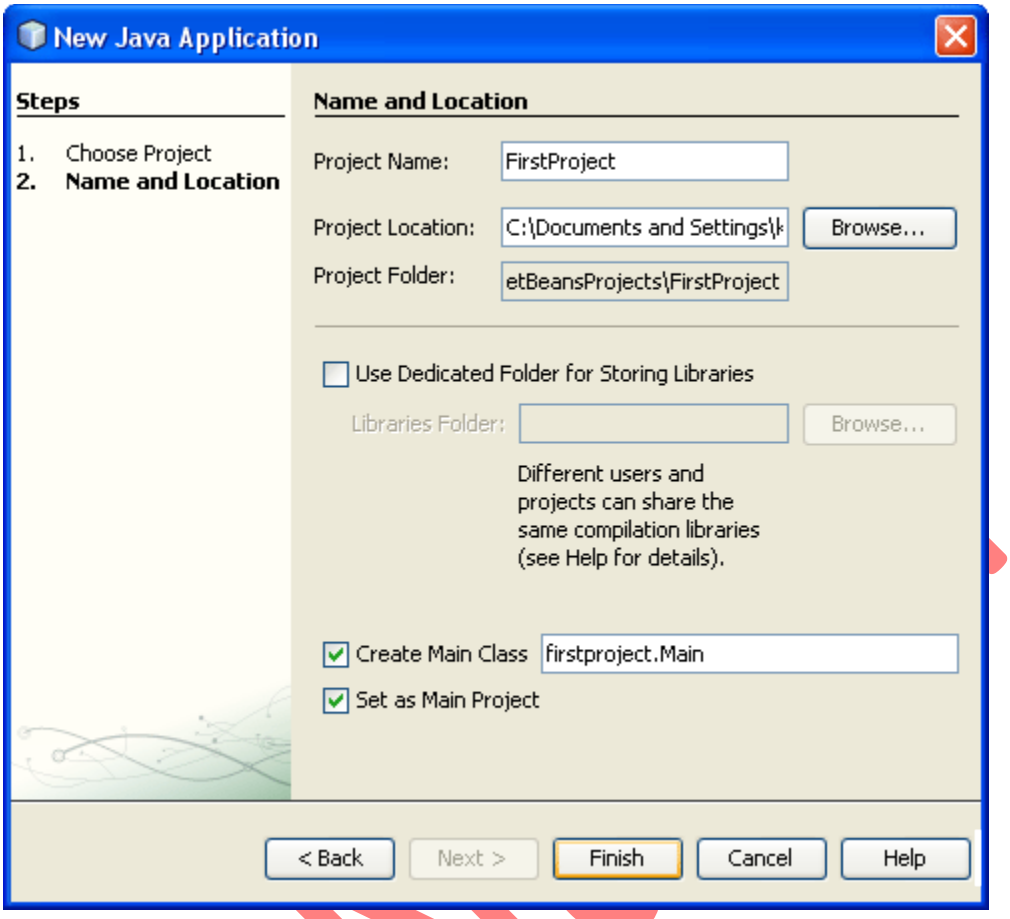
The NetBeans Software - First Start (38K)

You may have to drum your fingers and wait a while, as it's not the fastest thing in the world.

To start a new project, click on **File > New Project** from the NetBeans menu at the top. You'll see the following dialogue box appear:



We're going to be create a Java Application, so select **Java** under Categories, and then **Java Application** under Projects. Click the Next button at the bottom to go to step two:



In the Project Name area at the top, type a Name for your Project. Notice how the text at the bottom changes to match your project name (in the text box to the right of Create Main Class):

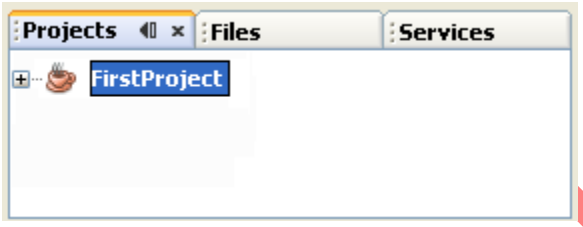
firstproject.Main

If we leave it like that, the Class will have the name Main. Change it to **FirstProject**:

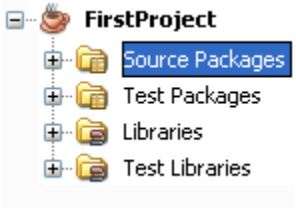
Now, the Class created will be called FirstProject, with a capital "F", capital "P". The package is also called firstproject, but with a lowercase "f" and lowercase "j".

The default location to save your projects appears in the Project Location text box. You can change this, if you prefer. NetBeans will also create a folder with your project name, in the same location. Click the Finish button and NetBeans will go to work creating all the necessary files for you.

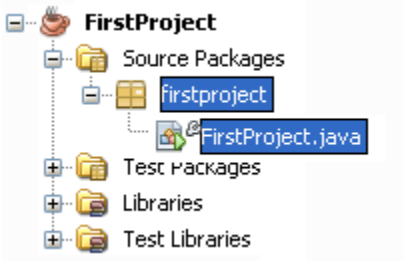
When NetBeans returns you to the IDE, have a look at the Projects area in the top left of the screen (if you can't see this, click **Window > Projects** from the menu bar at the top of the software):



Click the plus symbol to expand your project, and you'll see the following:

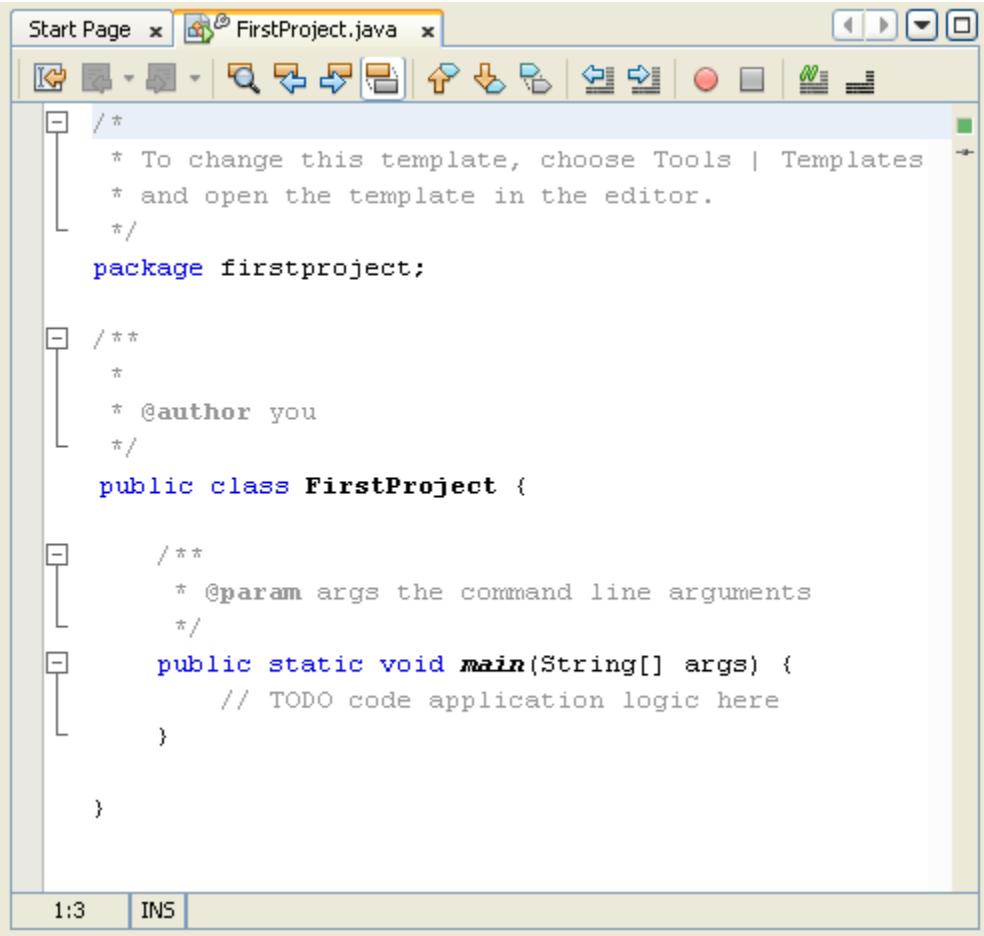


Now expand Source Packages to see your project name again. Expand this and you'll see the Java file that is your source code.



This same source code should be displayed to the right, in the large text area. It will be called **FirstProject.java**. If you can't see a code window, simply double click FirstProject.java in your Projects window above. The code will appear, ready for you to start work.

The coding window that appears should look like this (we've changed the author's name):



One thing to note here is that the class is called FirstProject:

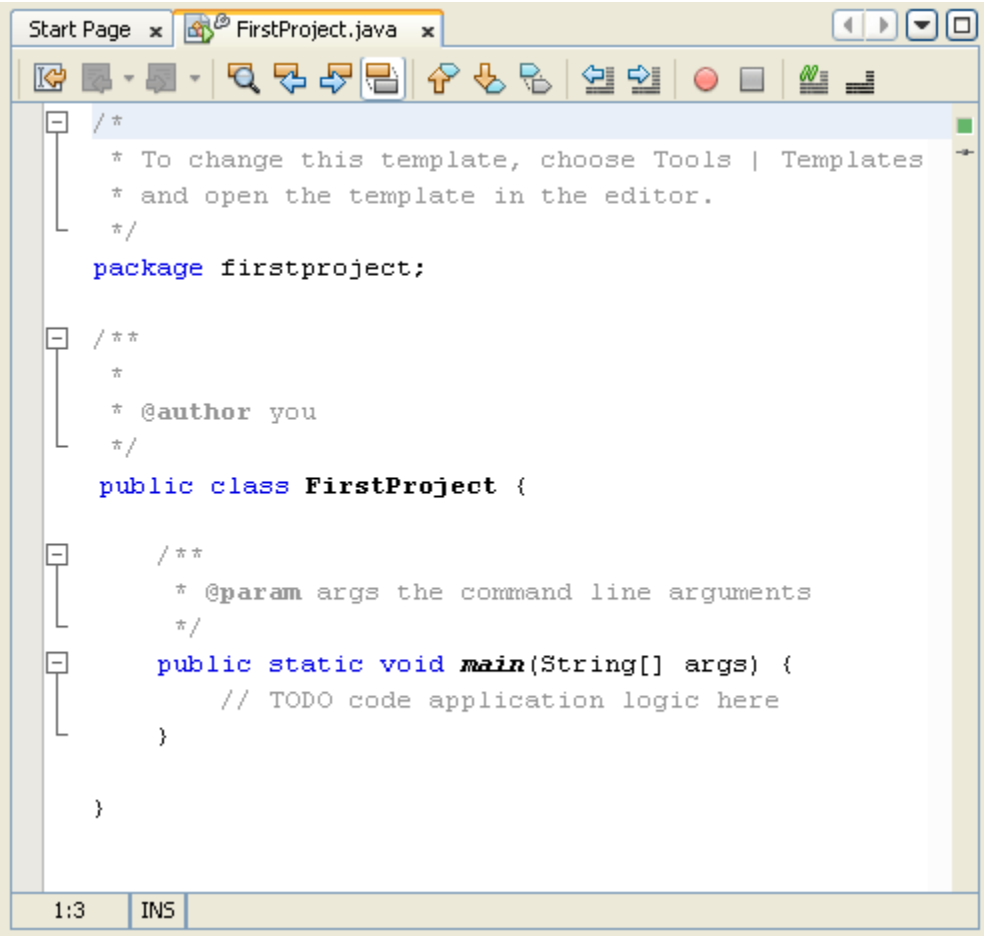
```
public class FirstProject {
```

This is the same name as the java source file in the project window: FirstProject.java. When you run your programmes, the compiler demands that the source file and the class name match. So if your .java file is called firstProject but the class is called FirstProject then you'll get an error on compile. And all because the first one is lowercase "f" and the second one uppercase.

Note that although we've also called the package firsproject, this is not necessary. We could have called the package someprogramme. So the name of the package doesn't have to be the same as the java source file, or the class in the source file: it's just the name of the java source file and the name of the class that must match.

Java Comments

When you create a New Project in NetBeans, you'll notice that some text is greyed out, with lots of slashes and asterisks:



The greyed-out areas are comments. When the programme runs, comments are ignored. So you can type whatever you want inside of your comments. But it's usual to have comments that explain what is you're trying to do. You can have a single line comment by typing two slashes, followed by your comment:

```
//This is a single line comment
```

If you want to have more than one line, you can either do this:

```
//This is a comment spreading
//over two lines or more
```

Or you can do this:

```
/*
This is a comment spreading
over two lines or more
*/
```

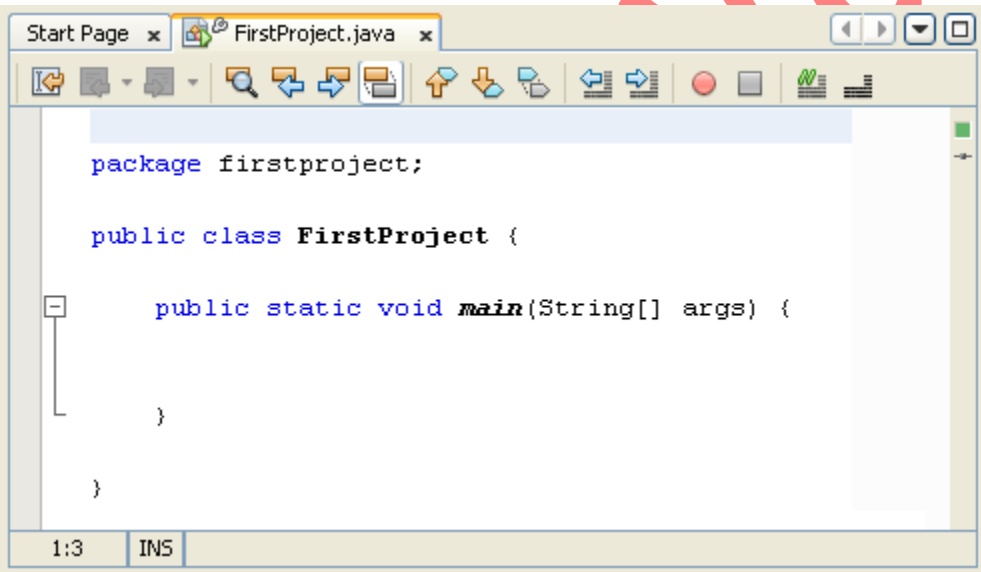
In the comment above, note how it starts with /*. To end the comment, we have */ instead.

There's also something called a Javadoc comment. You can see two of these in the coding image on the previous page. A Javadoc comment starts with a single forward slash and two asterisks (/**) and ends with an asterisk and one slash (*/). Each line of the comment starts with one asterisk:

```
/**
*This is a Javadoc comment
*/
```

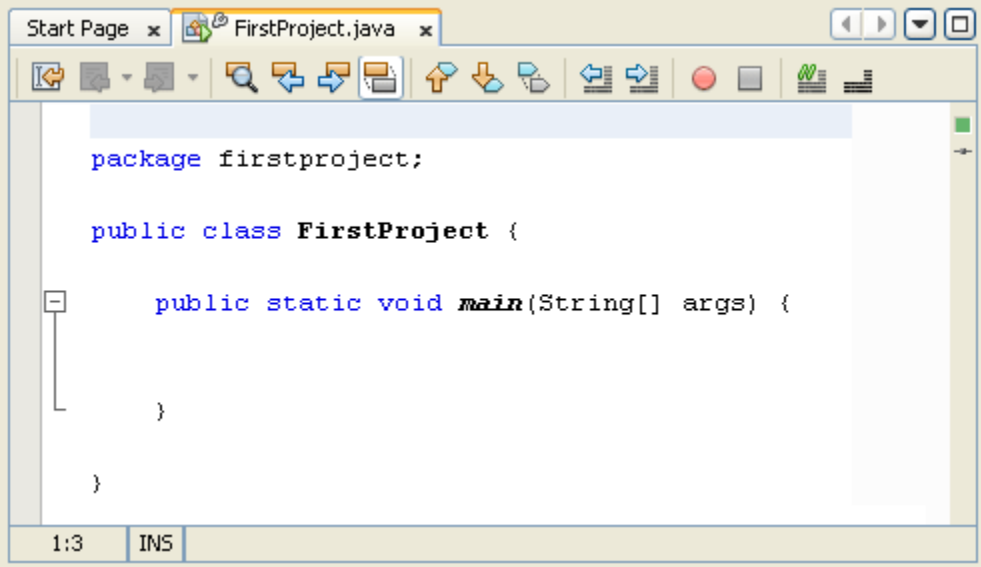
Javadoc comments are used to document code. The documented code can then be turned into an HTML page that will be helpful to others. You can see what these look like by clicking Run from the menu at the top of NetBeans. From the Run menu, select Generate Javadoc. There's not much to see, however, as you haven't written any code yet!

At this stage of your programming career, you can delete the comments that NetBeans generates for you. Here's our code again with the comments deleted:



The Structure of Java Code

Here's what your coding window should look like now:



You can see we have the package name first. Notice how the line ends with a semicolon. If you miss the semicolon out, the programme won't compile:

package **firstproject;**

The class name comes next:

public class **FirstProject {**

}

You can think of a class as a code segment. But you have to tell Java where code segments start and end. You do this with curly brackets. The start of a code segment is done with a left curly bracket { and is ended with a right curly bracket }. Anything inside of the left and right curly brackets belong to that code segment.

What's inside of the left and right curly brackets for the class is another code segment. This one:

public static void ***main*****(** String[ ] args **) {**

}

The word "main" is the important part here. Whenever a Java programme starts, it looks for a method called main. (A method is just a chunk of code. You'll learn more about these later.) It

then executes any code within the curly brackets for main. You'll get error messages if you don't have a main method in your Java programmes. But as its name suggest, it is the main entry point for your programmes.

The blue parts before the word "main" can be ignored for now.

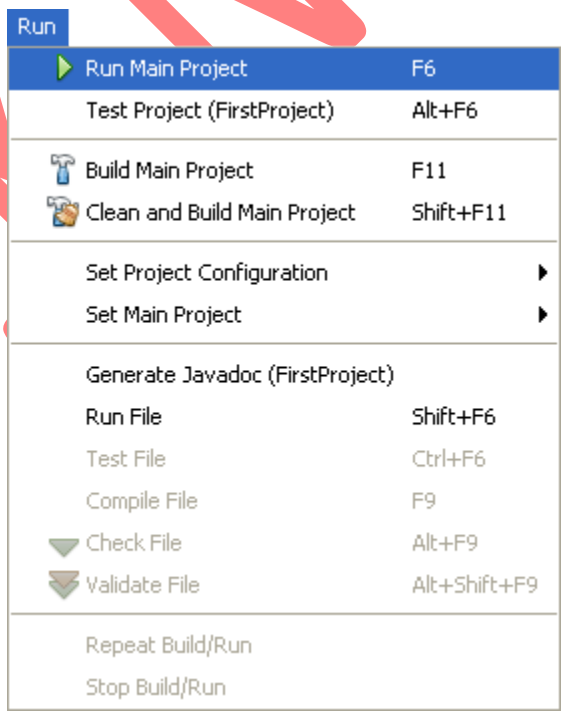
(If you're curious, however, public means that the method can be seen outside of this class; static means that you don't have to create a new object; and void means it doesn't return a value - it just gets on with it. The parts between the round brackets of main are something called command line arguments. Don't worry if you don't understand any of that, though.)

The important point to remember is that we have a class called FirstProject. This class contains a method called main. The two have their own sets of curly brackets. But the main chunk of code belongs to the class FirstProject.

Running your Java Programmes

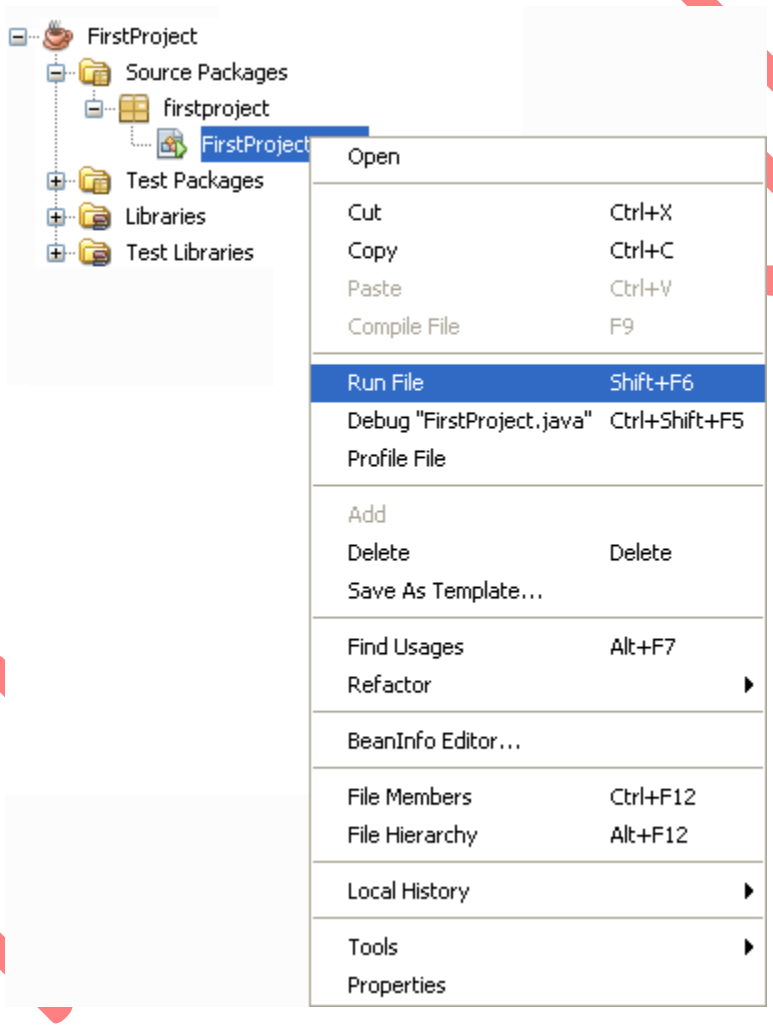
When you run a programme in NetBeans, it will run in the Output window at the bottom of your screen, just underneath your code. This is so that you don't have to start a terminal or console window - the Output window IS the console.

There are various ways to run your programme in NetBeans. The easiest way is to press F6 on your Keyboard. You can also run programmes using the menus as the top of NetBeans. Locate the **Run** menu, then select **Run Main Programme**:

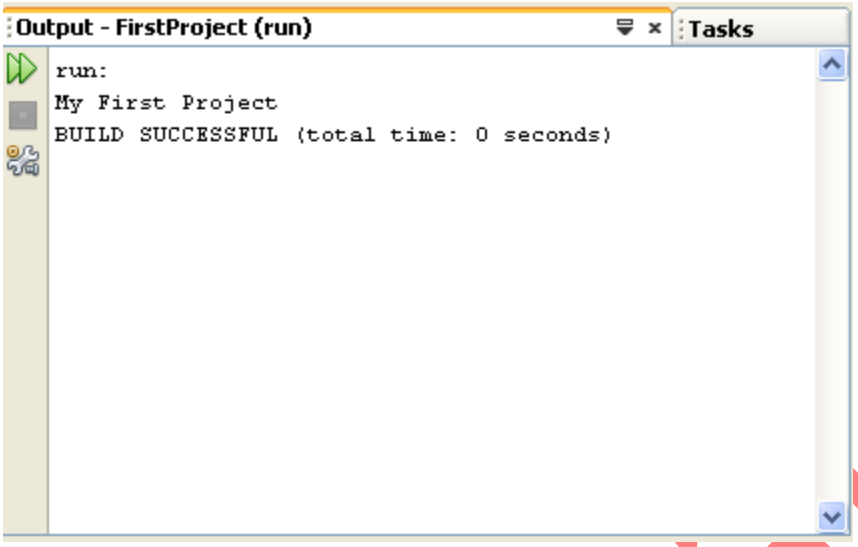


You can also click the green arrow on the NetBeans toolbar:

Another way to run your programmes is from the Projects window. This will ensure that the right source code is being run. Simply right click your java source file in the projects window and you'll see a menu appear. Select **Run File**.



Using one of the above methods, run your programme. You should see something happening in the Output window:



The second line in the Output window above is our code: My First Project. You can quickly run it again by clicking the two green arrows in the top left of the Output window.

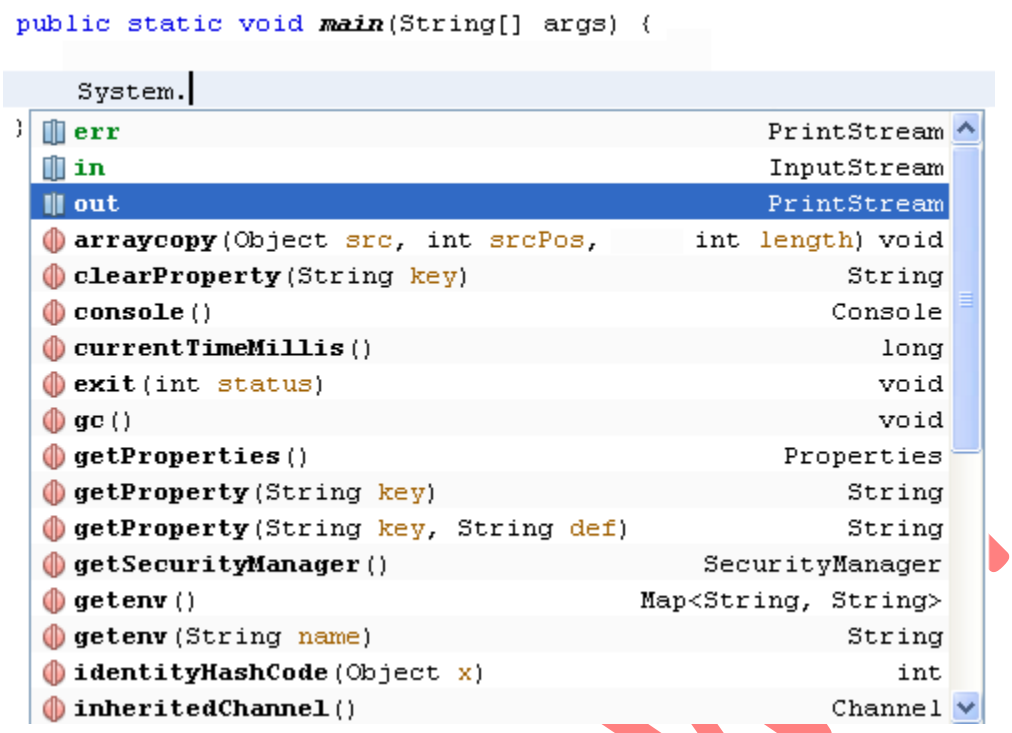
Printing to the Output Window

You can run the code you have so far, and turn it into a programme. It doesn't do anything, but it will still compile. So let's add one line of code just so that we can see how it works.

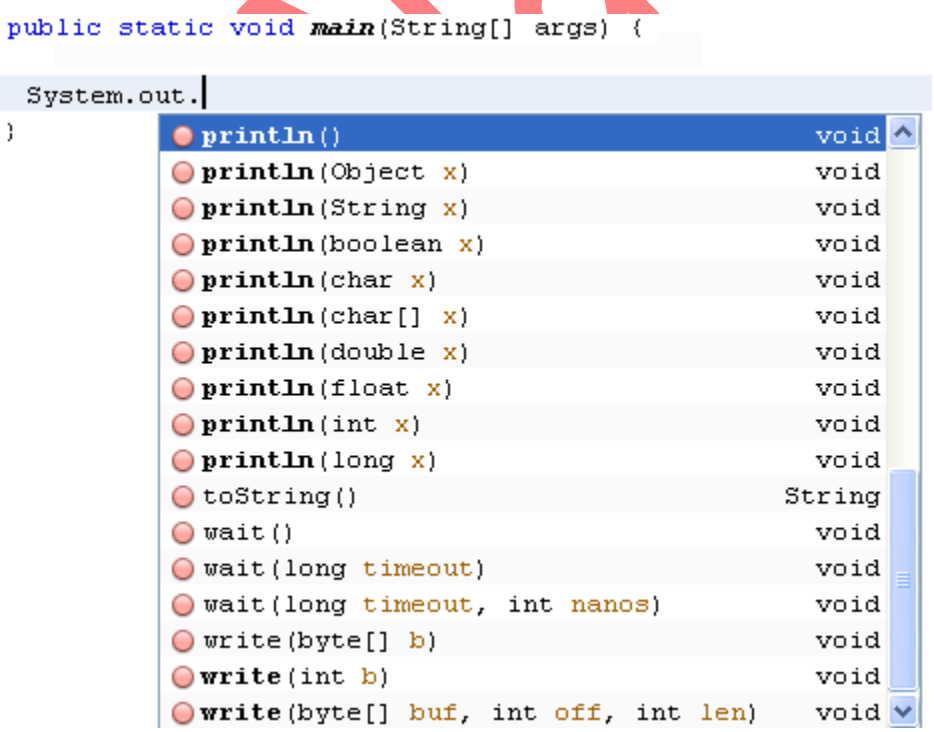
We'll output some text to a console window. Add the following line to your main method:

```
public static void main( String[ ] args ) {

System.out.println( "My First Project" );

}
```

When you type the full stop after "System", NetBeans will try to help you by displaying a list of available options:



Double click out to add it to your code, then type another full stop. Again, the list of options appears:



Select **println**( ). What this does is to print a line of text to the output screen. But you need to place your text between the round brackets of **println**. Your text needs to go between a pair of double quotes:

```
public static void main(String[] args) {

    System.out.println("");
}
```

Once you have your double quotes in place, type your text:

```
public static void main(String[] args) {

    System.out.println("My First Project");
}
```

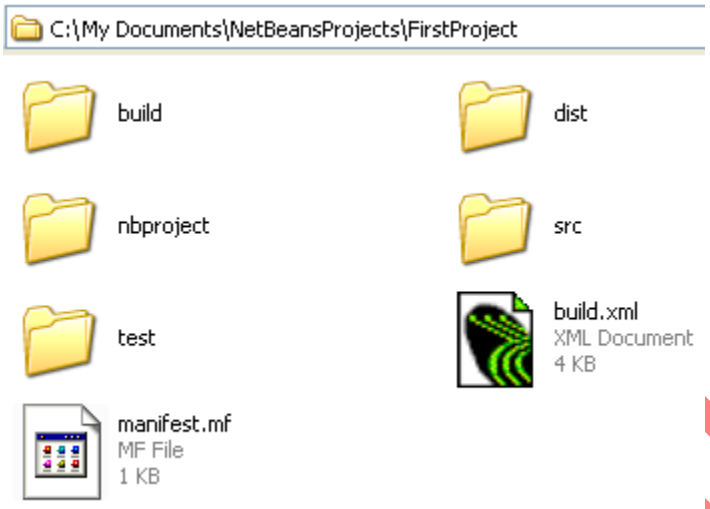
Notice that the line ends in a semicolon. Each complete line of code in Java needs a semicolon at the end. Miss it out and the programme won't compile.

OK, we can now go ahead and test this programme out. First, though, save your work. You can click **File** > **Save**, or **File** > **Save All**. Or click the **Save** icon on the NetBeans toolbar.

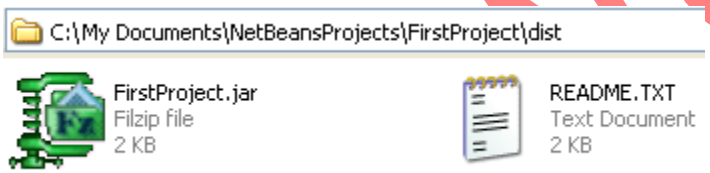
Sharing your Java Programmes

You can send your programmes to other people so that they can run them. To do that, you need to create a JAR file (Java Archive). NetBeans can do all this for you. From the **Run** menu at the top, select **Clean and Build Main Project**.

When you do, NetBeans saves your work and then creates all the necessary files. It will create a folder called dist and place all the files in there. Have a look in the place where your NetBeans projects are and you'll see the **dist** folder:



Double click the dist folder to see what's inside of it:



You should see a JAR file and README text file. The text file contains instructions on how to run the programme from a terminal/console window.

Java Variables

Programmes work by manipulating data placed in memory. The data can be numbers, text, objects, pointers to other memory areas, and more besides. The data is given a name, so that it can be re-called whenever it is need. The name, and its value, is known as a Variable. We'll start with number values.

To store a number in java, you have lots of options. Whole numbers such as 8, 10, 12, etc, are stored using the int variable. (The int stands for integer.) Floating point numbers like 8.4, 10.5, 12.8, etc, are stored using the double variable. You do the storing with an equals sign (=). Let's look at some examples (You can use your FirstProject code for these examples).

```
public static void main(String[ ] args) {

int first_number;

System.out.println("My First Project");

}
```

So to tell Java that you want to store a whole number, you first type the word int, followed by a space. You then need to come up with a name for your integer variable. You can call them almost anything you like, but there are a few rules:

- Variable names can't start with a number. So first_number is OK, but not 1st_number. You can have numbers elsewhere in the variable name, just not at the start.
- Variable names can't be the same as Java keywords. There are quite a lot of these, and they will turn blue in NetBeans, like **int** above.
- You can't have spaces in your variable names. The variable declaration **int first number** will get you an error. We've used the underscore character, but it's common practise to have the first word start with a lowercase letter and the second or subsequent words in uppercase: firstNumber, myFirstNumber
- Variable names are case sensitive. So firstNumber and FirstNumber are different variable names.

To store something in the variable called **first_number**, you type an equals sign and then the value you want to store:

```
public static void main(String[ ] args) {

int first_number;

first_number = 10;

System.out.println("My First Project");

}
```

So this tells java that we want to store a value of 10 in the integer variable that we've called first_number.

If you prefer, you can do all this on one line:

```
public static void main(String[ ] args) {

int first_number = 10;

System.out.println("My First Project");

}
```

To see all this in action, change the println method slightly to this:

```
System.out.println( "First number = " + first_number );
```

What we now have between the round brackets of println is some direct text enclosed in double quotes:

("First number = "

We also have a plus sign, followed by the name of our variable:

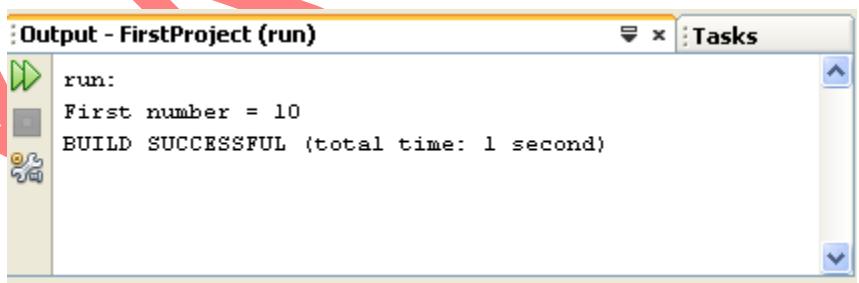
+ first_number);

The plus sign means "join together". So we're joining together the direct text and the name of our variable. The joining together is known as concatenation.

Your coding window should now look like this (note how each line of code ends with a semicolon):

```
package firstproject;

public class FirstProject {

  public static void main(String[] args) {

      int first_number;
      first_number = 10;

      System.out.println("First number = " + first_number );

  }

}
```

Run your programme and you should see the following in the Output window at the bottom:



So the number that we stored in the variable called **first_number** is output after the equals sign.

Let's try some simple addition. Add two more **int** variables to your code, one to store a second number, and one to store an answer:

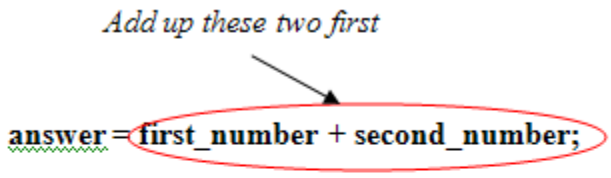
int **first_number, second_number, answer;**

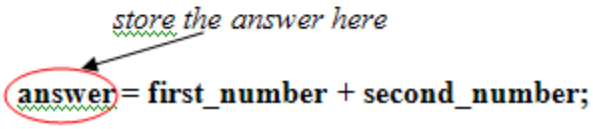
Notice how we have three variable names on the same line. You can do this in Java, if the variables are of the same type (the **int** type, for us). Each variable name is then separated by a comma.

We can then store something in the new variables:

```
first_number = 10;
second_number = 20;
answer = first_number + second_number;
```

For the answer variable, we want to add the first number to the second number. Addition is done with the plus (+) symbol. Java will then add up the values in first_number and second_number. When it's finished it will store the total in the variable on the left of the equals sign. So instead of assigning 10 or 20 to the variable name, it will add up and then do the assigning. In case that's not clear, here's what happens:





The above is equivalent to this:

```
answer = 10 + 20;
```

But Java already knows what is inside of the two variables, first_number and second_number, so you can just use the names.

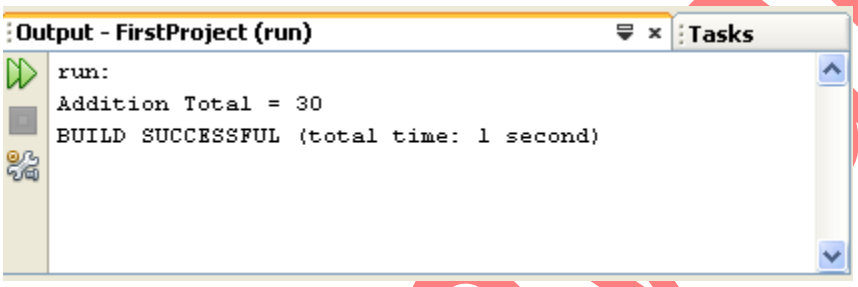
Now change your println method slightly to this:

```
System.out.println("Addition Total = " + answer );
```

Again, we are combining direct text surrounded by double quotes with a variable name. Your coding window should look like this:

```
public static void main(String[] args) {

    int first_number, second_number, answer;

    first_number = 10;
    second_number = 20;
    answer = first_number + second_number;

    System.out.println("Addition Total = " + answer );
}
```

When you run your programme you should get the following in the output window:



So our programme has done the following:

- Stored one number
- Stored a second number
- Added these two numbers together
- Stored the result of the addition in a third variable
- Printed out the result

You can also use numbers directly. Change the answer line to this:

answer = first_number + second_number + 12;

Run your programme again. What printed out? Was it what you expected?

You can store quite large numbers in the **int** variable type. The maximum value is 2147483647. If you want a minus number the lowest value you can have is -2147483648. If you want larger or smaller numbers you can use another number variable type: **double**.

The Double Variable

The double variable can hold very large (or small) numbers. The maximum and minimum values are 17 followed by 307 zeros.

The double variable is also used to hold floating point values. A floating point value is one like 8.7, 12.5, 10.1. In other words, it has a "point something" at the end. If you try to store a floating point value in an int variable, NetBeans will underline the faulty code. If you try to run the programme, the compiler will throw up an error message.

Let's get some practise using doubles.

Change the **int** from your previous code to double. So change this:

int **first_number, second_number, answer;**

to this:

double **first_number, second_number, answer;**

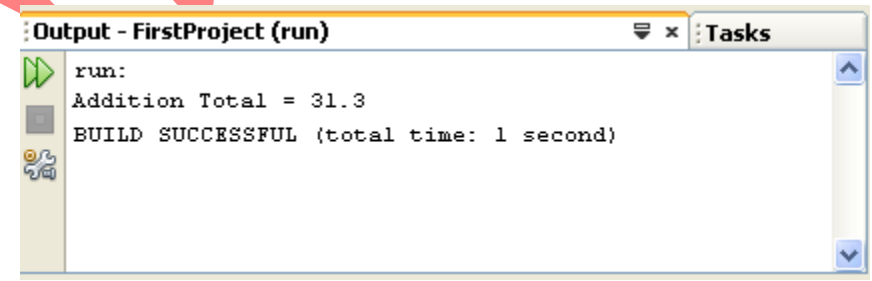
Now change the values being stored:

first_number = 10.5;
second_number = 20.8;

Leave the rest of the programme as it is. Your coding window should look like this:

```
public static void main(String[] args) {

    double first_number, second_number, answer;

    first_number = 10.5;
    second_number = 20.8;
    answer = first_number + second_number;

    System.out.println("Addition Total = " + answer );
}
```

Run your programme again. The output window should look like this:



Try changing the values stored in first_number and second_number. Use any values you like. Run your programme and see what happens.

Short and Float Variables

Two more variable types you can use are **short** and **float**. The **short** variable type is used to store smaller number, and its range is between minus 32,768 and plus 32,767. Instead of using **int** in our code on the previous pages, we could have used short instead. You should only use short if you're sure that the values that you want to hold don't go above 32, 767 or below -32,768.

The double value we used can store really big numbers of the floating point variety. Instead of using double, **float** can be used. When storing a value in a float variable, you need the letter "f" at the end. Like this:

```
float first_number, second_number, answer;

first_number = 10.5f;
second_number = 20.8f;
```

So the letter "f" goes after the number but before the semicolon at the end of the line. To see the difference between float and double, see below.

Simple Arithmetic

With the variables you've been using, you can use the following symbols to do calculations:

+ (the plus sign)
- (the minus sign)
* (the multiplication sign is the asterisk sign)
/ (the divide sign is the forward slash)

Try this exercise:

Delete the plus symbol that is used to add first_number and second_number. Replace it with the symbols above, first the minus sign, then the multiplication sign, and then the divide. The answer to the final one, the divide, should give you a really big number.

The number you should get for divide is 0.5048076923076923. This is because you used the double variable type. However, change the double to float. Then add the letter "f" to the end of the numbers. So your code should look like this:

```
public static void main(String[] args) {

    float first_number, second_number, answer;

    first_number = 10.5f;
    second_number = 20.8f;
    answer = first_number / second_number;

    System.out.println("Total = " + answer );
}
```

When you run the above code, the answer is now 0.5048077. Java has taken the first 6 numbers after the point and then rounded up the rest. So the double variable type can hold more numbers than float. (Double is a 64 bit number and float is only 32 bit.)

Operator Precedence

You can, of course, calculate using more than two numbers in Java. But you need to take care of what exactly is being calculated. Take the following as an example:

first_number = 100;
second_number = 75;
third_number = 25;

answer = first_number - second_number + third_number;

If you did the calculation left to right it would be 100 - 75, which is 25. Then add the third number, which is 25. The total would be 50. However, what if you didn't mean that? What if you wanted to add the second and third numbers together, and then deduct the total from the first number? So 75 + 25, which is 100. Then deduct that from the first number, which is 100. The total would now be 0.

To ensure that Java is doing what you want, you can use round brackets. So the first calculation would be:

answer = (first_number - second_number) + third_number;

Here's the coding window so that you can try it out:

```
public static void main(String[] args) {

    int first_number, second_number, third_number, answer;

    first_number = 100;
    second_number = 75;
    third_number = 25;
    answer = (first_number - second_number) + third_number;

    System.out.println("Total = " + answer );
}
```

The second calculation is this:

answer = first_number - (second_number + third_number);

And here's the code window:

```
public static void main(String[] args) {

    int first_number, second_number, third_number, answer;

    first_number = 100;
    second_number = 75;
    third_number = 25;
    answer = first_number - (second_number + third_number);

    System.out.println("Total = " + answer );
}
```

Now let's try some multiplication and addition.

Change your math symbols (called Operators) to plus and multiply:

answer = first_number + second_number * third_number;

Delete all your round brackets, and then run your programme.

With no brackets, you'd think Java would calculate from left to right. So you'd think it would add the first number to the second number to get 175. Then you'd think it would multiply by the third number, which is 25. So the answer would be 4375. Run the programme, though. The answer that you actually get is 1975! So what's going on?

The reason Java got the "wrong" answer was because of Operator Precedence. Java treats some mathematical symbols as more important than others. It sees multiplication as having a priority over addition, so it does this first. It then does the addition. So Java is doing this:

answer = first_number + (second_number * third_number);

With the round brackets in place, you can see that second number is being multiplied by third number. The total is then added to the first number. So 75 multiplied by 25 is 1875. Add 100 and you get 1975.

If you want it the other way round, don't forget to "tell" Java by using round brackets:

answer = (first_number + second_number) * third_number;

Division is similar to multiplication: Java does the dividing first, then the addition or subtraction. Change your answer line to the following:

answer = first_number + second_number / third_number;

The answer you get is 103. Now add some round brackets:

answer = (first_number + second_number) / third_number;

The answer this time will be 7. So without the round brackets, Java does the dividing first, and then adds 100 to the total - it doesn't work from left to right.

Here's a list on Operator Precedence

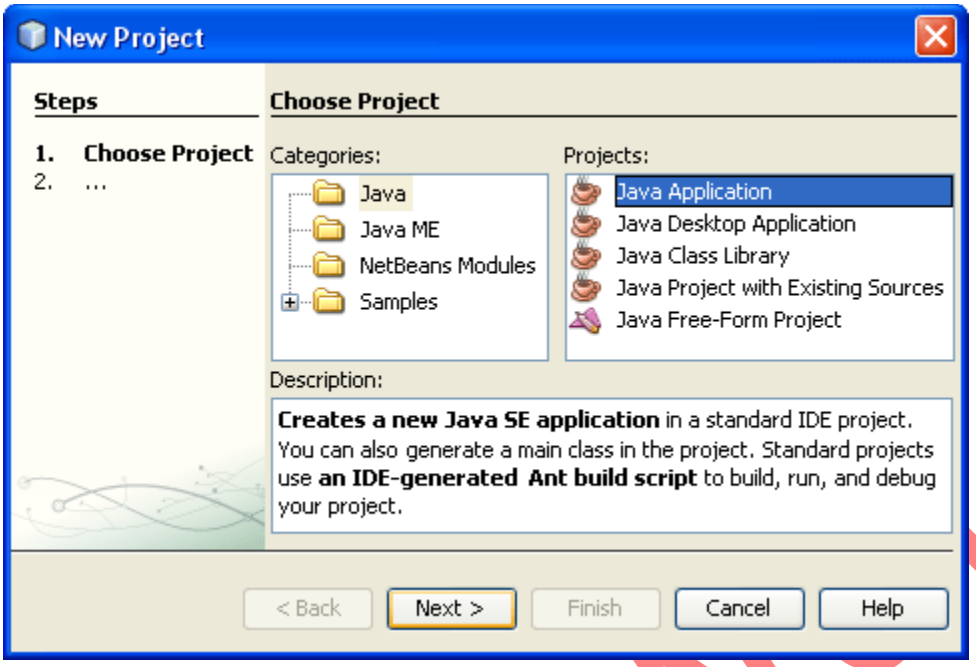
- Multiply and Divide - Treated equally, but have priority over Addition and Subtraction
- Add and Subtract - Treated equally but have a lower priority than multiplication and division

So if you think Java is giving you the wrong answer, remember that Operator Precedence is important, and add some round brackets.

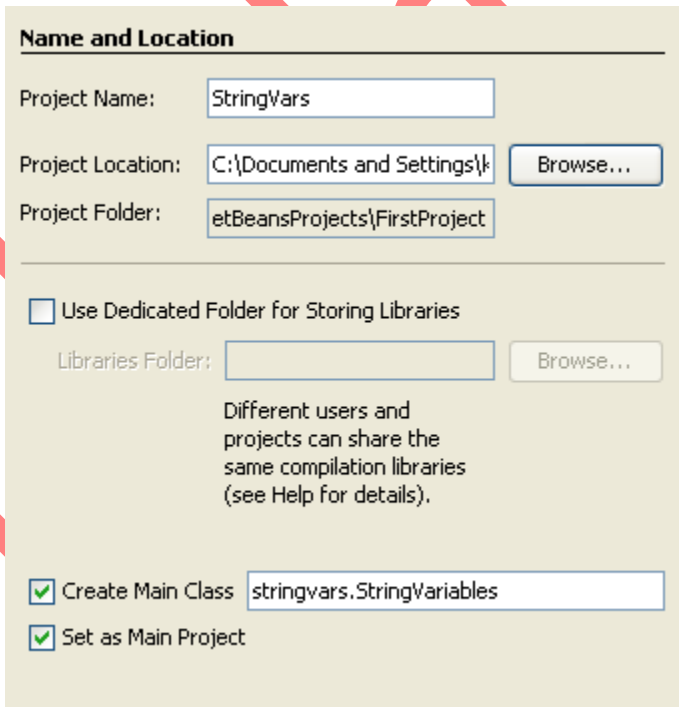
String Variables

As well as storing number values, variables can hold text. You can store just one character, or lots of characters. To store just one character, the char variable is used. Usually, though, you'll want to store more than one character. To do so, you need the string variable type.

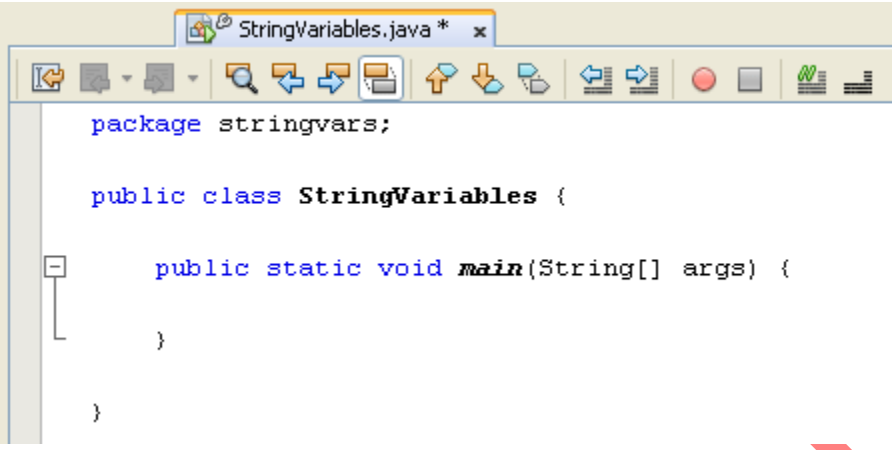
Start a new project for this by clicking **File > New Project** from the menu bar at the top of NetBeans. When the **New Project** dialogue box appears, make sure **Java** and **Java Application are selected**:



Click **Next** and type **StringVars** as the Project Name. Make sure there is a tick in the box for **Create Main Class**. Then delete **Main** after **stringvars**, and type **StringVariables** instead, as in the following image:



So the Project Name is **StringVars**, and the Class name is **StringVariables**. Click the **Finish** button and your coding window will look like this (we've deleted all the default comments). Notice how the package name is all lowercase (stringvars) but the Project name was StringVars.



To set up a string variable, you type the word **String** followed by a name for your variable. Note that there's an uppercase "S" for String. Again, a semicolon ends the line:

String **first_name**;

Assign a value to your new string variable by typing an equals sign. After the equals sign the text you want to store goes between two sets of double quotes:

first_name = "William";

If you prefer, you can have all that on one line:

String **first_name = "William";**

Set up a second string variable to hold a surname/family name:

String **family_name = "Shakespeare";**

To print both names, add the following println( ):

System.out.println(first_name + " " + family_name **);**

In between the round brackets of **println**, we have this:

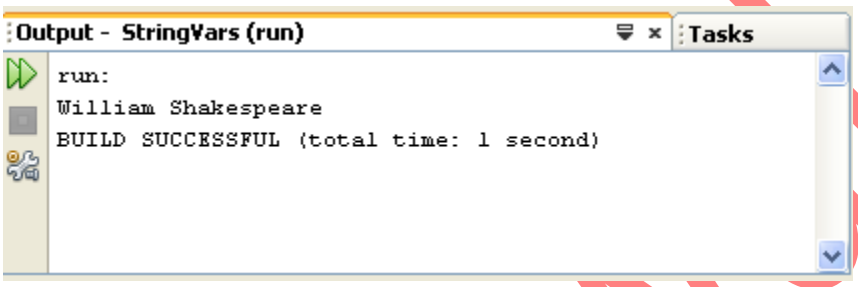
first_name + " " + family_name

We're saying print out whatever is in the variable called **first_name**. We then have a plus symbol, followed by a space. The space is enclosed in double quotes. This is so that Java will recognise that we want to print out a space character. After the space, we have another plus symbol, followed by the **family_name** variable.

Although this looks a bit messy, we're only printing out a first name, a space, then the family name. Your code window should look like this:

```
public static void main(String[] args) {

    String first_name = "William";
    String family_name = "Shakespeare";

    System.out.println(first_name + " " + family_name);
}
```

Run your programme and you should see this in the Output window:



If you are storing just a single character, then the variable you need is **char** (lowercase "c"). To store the character, you use single quotes instead of double quotes. Here's our programme again, but this time with the char variable:

```
public static void main(String[] args) {

    char first_name = 'W';
    char family_name = 'S';

    System.out.println(first_name + " " + family_name);
}
```

If you try to surround a char variable with double quotes, NetBeans will underline the offending code in red, giving you "incompatible type" errors. You can, however, have a String variable with just a single character. But you need double quotes. So this is OK:

String first_name = "W";

But this is not:

String first_name = 'W';

The second version has single quotes, while the first has double quotes.

Accepting Input from a User

One of the strengths of Java is the huge libraries of code available to you. This is code that has been written to do specific jobs. All you need to do is to reference which library you want to use, and then call a method into action. One really useful class that handles input from a user is called the **Scanner** class. The Scanner class can be found in the **java.util** library. To use the Scanner class, you need to reference it in your code. This is done with the keyword **import**.

```
import java.util.Scanner;
```

The import statement needs to go just above the Class statement:

```
import java.util.Scanner;

public class StringVariables {


}
```

This tells java that you want to use a particular class in a particular library - the Scanner class, which is located in the java.util library.

The next thing you need to do is to create an object from the Scanner class. (A class is just a bunch of code. It doesn't do anything until you create a new object from it.)

To create a new Scanner object the code is this:

```
Scanner user_input = new Scanner( System.in );
```

So instead of setting up an **int** variable or a **String** variable, we're setting up a **Scanner** variable. We've called ours **user_input**. After an equals sign, we have the keyword **new**. This is used to create new objects from a class. The object we're creating is from the Scanner class. In between round brackets we have to tell java that this will be System Input (System.in).

To get the user input, you can call into action one of the many methods available to your new Scanner object. One of these methods is called **next**. This gets the next string of text that a user types on the keyboard:

```
String first_name;
first_name = user_input.next( );
```

So after our user_input object we type a dot. You'll then see a popup list of available methods. Double click **next** and then type a semicolon to end the line. We can also print some text to prompt the user:

```
String first_name;
System.out.print("Enter your first name: ");
first_name = user_input.next( );
```

Notice that we've used **print** rather than **println** like last time. The difference between the two is that println will move the cursor to a new line after the output, but print stays on the same line.

We'll add a prompt for a family name, as well:

```
String family_name;
System.out.print("Enter your family name: ");
family_name = user_input.next( );
```

This is the same code, except that java will now store whatever the user types into our family_name variable instead of our first_name variable.

To print out the input, we can add the following:

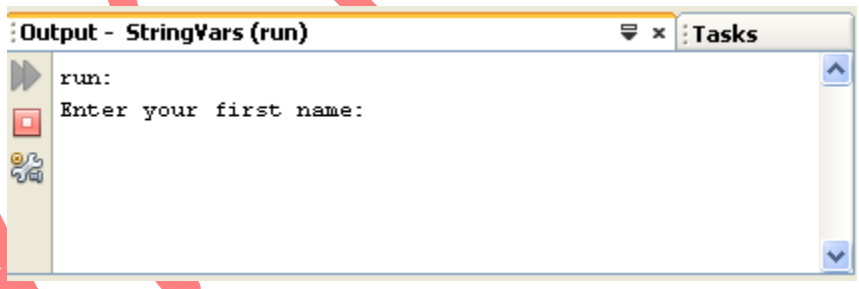
```
String full_name;
full_name = first_name + " " + family_name;

System.out.println("You are " + full_name);
```

We've set up another String variable, **full_name**. We're storing whatever is in the two variables **first_name** and **family_name**. In between the two, we've added a space. The final line prints it all out in the Output window.

So adapt your code so that it matches that in the next image:

```
package stringvars;

import java.util.Scanner;

public class StringVariables {

    public static void main(String[] args) {

        Scanner user_input = new Scanner(System.in);

        String first_name;
        System.out.print("Enter your first name: ");
        first_name = user_input.next();

        String family_name;
        System.out.print("Enter your family name: ");
        family_name = user_input.next();

        String full_name;
        full_name = first_name + " " + family_name;

        System.out.println("You are " + full_name);

    }
}
```

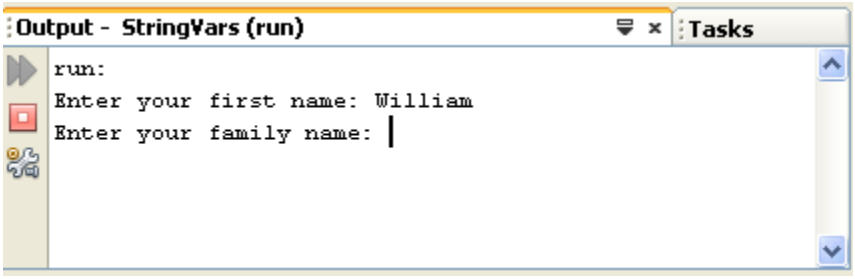
Run your programme until your Output window displays the following:



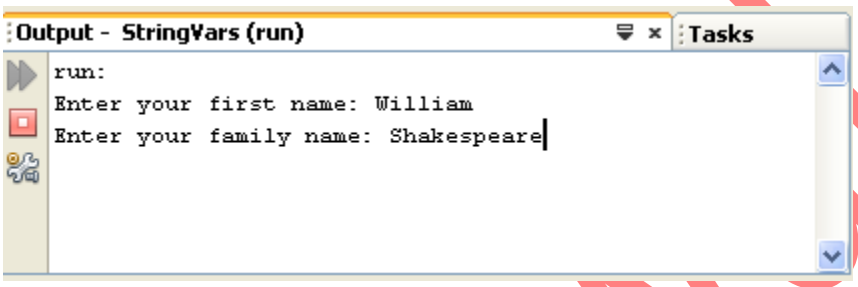
Java is now pausing until you enter something on your keyboard. It won't progress until you hit the enter key. So left click after "Enter your first name:" and you'll see your cursor flashing away. Type a first name, and then hit the enter key on your keyboard.

After you hit the enter key, java will take whatever was typed and store it in the variable name to the left of the equals sign. For us, this was the variable called first_name.

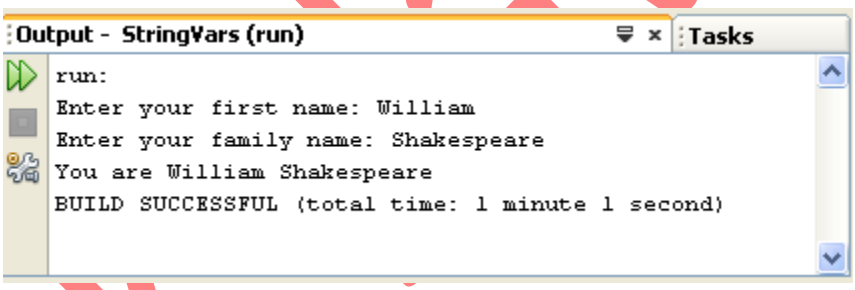
The programme then moves on to the next line of code:



Type a family name, and hit the enter key again:



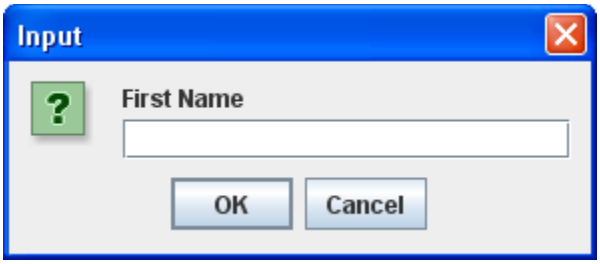
The user input has now finished, and the rest of the programme executes. This is the output of the two names. The final result should like this:



So we used the Scanner class to get input from a user. Whatever was typed was stored in variables. The result was then printed to the Output window.

Java Option Panes

Another useful class for accepting user input, and displaying results, is the JOptionPane class. This is located in the javax.swing library. The JOptionPane class allows you to have input boxes like this one:



And message boxes like this:

Let's adapt our code from the previous section and have some option panes.

The first thing to do is to reference the library we want to use:

import javax.swing.JOptionPane;

This tells java that we want to use the JOptionPane class, located in the javax.swing library.

You can start a new project for this, if you prefer not to adapt your previous code. (You should know how to create a new project by now. Just remember to change the name of the Class from Main to something else. We're going to have the class name InputBoxes for ours. Our package name will be userinput.)

Add the import line to your new project, and your code window should look like something like this:

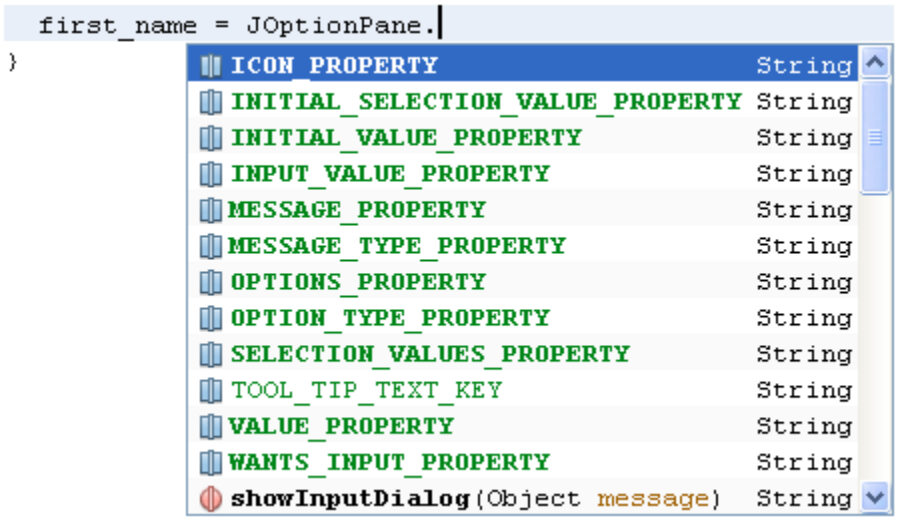
```
package userinput;
import javax.swing.JOptionPane;

public class InputBoxes {

    public static void main(String[] args) {

    }

}
```

(The reason for the wavy underline is that we haven't used the JOptionPane class yet. It will go away once we do.)

To get an input box that the user can type into, we can use the **showInputDialog** method of JOptionPane. We'll store the input straight into a first name variable again, just like last time. So add the following line to your main method:

```
String first_name;
first_name = JOptionPane.showInputDialog("First Name");
```

As soon as you type a full stop after JOptionPane you will see the following popup list:



Double click **showInputDialog**. In between the round brackets of showInputDialog type the message that you want displayed above the input text box. We've typed "First name". Like all strings, it needs to go between double quotes.

Add the following code so that we can get the user's family name:

```
String family_name;
family_name = JOptionPane.showInputDialog("Family Name");
```

Join the two together, and add some text:

```
String full_name;
full_name = "You are " + first_name + " " + family_name;
```

To display the result in a message box, add the following:

```
JOptionPane.showMessageDialog( null, full_name );
```

This time, we want **showMessageDialog** from the popup list. In between the round brackets we first have the word **null**. This is a java keyword and just means that the message box is not

associated with anything else in the programme. After a comma comes the text we want to display in the message box. The whole of your code should look like this:

```
package userinput;
import javax.swing.JOptionPane;

public class InputBoxes {

    public static void main(String[] args) {

      String first_name;
      first_name = JOptionPane.showInputDialog("First Name");

      String family_name;
      family_name = JOptionPane.showInputDialog("Family Name");

      String full_name;
      full_name = "You are " + first_name + " " + family_name;

      JOptionPane.showMessageDialog(null, full_name);
      System.exit(0);

    }
}
```

Notice the line at the bottom of the code:

System.exit(0);

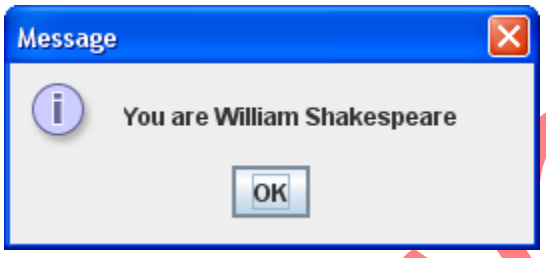
As its name suggests, this ensures that the programme exits. But it also tidies up for us, removing all the created objects from memory.

Now run your code. (Another quick way to run your programme in NetBeans is by right-clicking anywhere inside of the coding window. From the menu that appears, select **Run File**.)

You'll see the First Name input box. Type something into it, then click OK:

When the Family Name input box appears, type a family name and click OK:

After you click OK, the message box will display:



Click OK to end the programme.

Exercise
Input boxes and Message boxes can be formatted further. Try the following for your Input boxes:

showInputDialog("First Name", "Enter Your First Name");

showInputDialog("Family", "Enter Your Family Name");

Exercise
For your Message boxes try this (yours should be on one line):

showMessageDialog(null, full_name, "Name",
JOptionPane.INFORMATION_MESSAGE);

Exercise
Instead of JOptionPane.INFORMATION_MESSAGE try these:

ERROR_MESSAGE
PLAIN_MESSAGE
QUESTION_MESSAGE
WARNING_MESSAGE

Exercise
Input boxes are not just used for text: they can accept numbers as well. Write a programme that prompts the user for two numbers, the breadth of a rectangle and the height of a rectangle. Use a

message box to calculate the area of the rectangle. (Remember: the area of a rectangle is its breadth multiplied by the height.) However, you'll need some extra help for this exercise.

Help for the Exercise
You have to use the String variable to get your numbers from the user:

```
String breadth;
breadth = JOptionPane.showInputDialog("Rectangle Breadth");
```

However, you can't multiply two strings together. You need to convert the Strings to integers. You can convert a string to an integer like this:

```
Integer.parseInt( string_to_convert )
```

So you type Integer then a full stop. After the stop, type parseInt( ). In between the round brackets of parseInt, type the name of the string variable you're trying to convert.

Set up an int variable for the area. You can then multiply and assign on the same line;

```
int area = Integer.parseInt( string_one ) * Integer.parseInt( string_two);
```

For the message box, use concatenation:

```
"answer = " + area
```

You can use any of the MESSAGE symbols for your message box.

Exercise
The programme will crash if you enter floating point values for the breadth and height. How would you solve this?

When you have solved the above exercise, do you really want Integer.parseInt? What else do you think you can use?

Conditional Logic - If Statements

The programming you're doing now is sequential programming, meaning the code is executed from top to bottom. It's very linear, in that each and every line of code will be read, starting with the first line of code you write and ending at the last line.

But you don't always want your programmes to work like that. Often, you want code to be executed only if certain conditions are met. For example, you might want one message to display if a user is below the age of 18 and a different message if he or she is 18 or older. You want to control the flow of the programme for yourself. You can do this with conditional logic.

Conditional logic is mainly about the IF word: IF user is less than 18 then display this message; IF user is 18 or older then display that message. Fortunately, it's very easy to use conditional logic in Java. Let's start with IF Statements.

IF Statements

Executing code when one thing happens rather than something else is so common in programming that that the IF Statement has been developed. The structure of the IF Statement in Java is this:

```
if ( Statement ) {

}
```

You start with the word IF (in lowercase) and a pair of round brackets. You then use a pair of curly brackets to section off a chunk of code. This chunk of code is code that you only want to execute IF your condition is met. The condition itself goes between round brackets:

```
if ( user < 18 ) {

}
```

This condition says "IF user is less than 18". But instead of saying "is less than" we use the shorthand notation of the left-pointing angle bracket (<). IF the user is less than 18 then we want something to happen, to display a message, for example:

```
if ( user < 18 ) {

//DISPLAY MESSAGE

}
```

If the user is not less than 18 then the code between the curly brackets will be skipped, and the programme continues on its way, downwards towards the last line of code. Whatever you type between the curly brackets will only be executed IF the condition is met, and this condition goes between the round brackets.

Before we try this out, another shorthand notation is this symbol >. The right-pointing angle bracket means "greater than". Our IF Statement above can be amended slightly to check for users who are greater than 18:

```
if ( user > 18 ) {

//DISPLAY MESSAGE

}
```

The only thing new in this code is the > symbol. The condition now checks for users who are greater than 18.

But the condition doesn't check for people who are exactly 18, just those greater than 18. If you want to check for those who are 18 or over, you can say "greater than or equal to". The symbols for this are the greater than sign (>) followed by an equals sign (=):

```
if ( user >= 18 ) {

//DISPLAY MESSAGE

}
```

You can also check for "less than or equal to" in a similar way:

```
if ( user <= 18 ) {

//DISPLAY MESSAGE

}
```

The above code contains a less than symbol (<) followed by the equals sign.

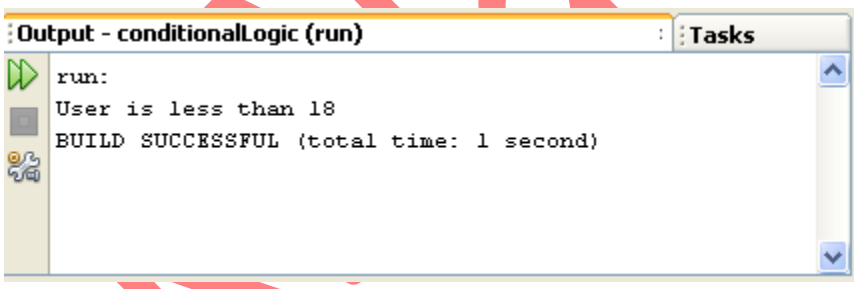
Let's try all this out in a simple programme.

Start a new project by clicking **File > New Project** from the menu bar in NetBeans. You can call your package and class names anything you like. Enter the following code (our package name is **conditionallogic** and the Class is called **IFStatements**):

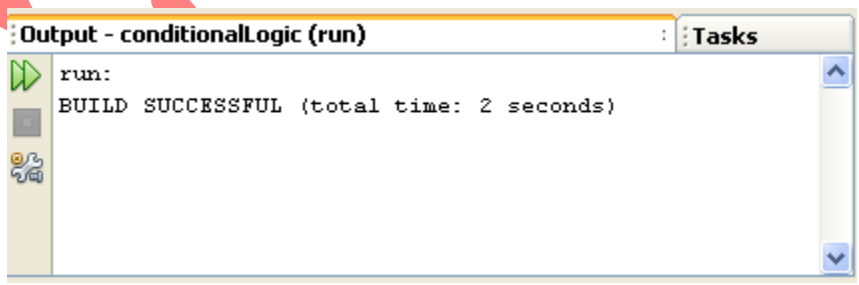
```
package conditionallogic;

public class IFStatements {

    public static void main(String[] args) {

        int user = 17;

        if (user < 18) {
            System.out.println("User is less than 18");
        }

    }
}
```

We've set up an integer variable, and assigned a value of 17 to it. The IF statement checks for "less than 18". So the message between the curly brackets should be printed out.

Run your programme and check it out. (NetBeans has a habit of running the programme in bold text in the Projects window and not the code you have displayed. To run the code in your coding window, right click anywhere in the code. From the menu that appears select Run File.) You should see this in your Output window:



Now change the value for the user variable from 17 to 18. Run your programme again. You should see this:



So the programme runs OK, with no error messages. It's just that nothing gets printed out. The reason is that the message code is between the curly brackets of the IF Statement. And the IF

Statement is checking for values less than 18. IF the condition is not met, Java ignores the curly brackets altogether and moves on.

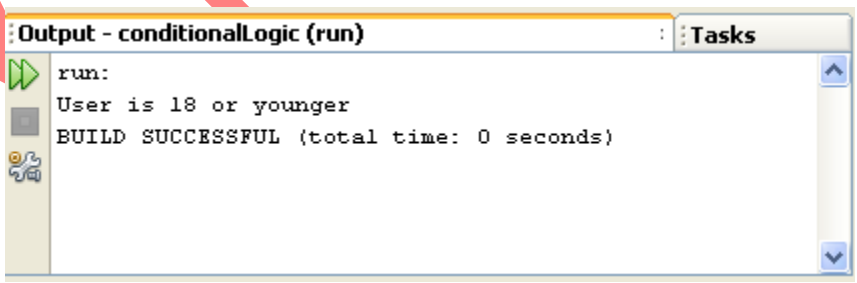
Exercise
Replace your "less than" symbol with the "less than or equal to" symbols. Change your message to suit, something like "user is less than or equal to 18". Run your programme again. Do you see the message?

Exercise
Change the user value to 20. Run your programme again. Do you still see the message?

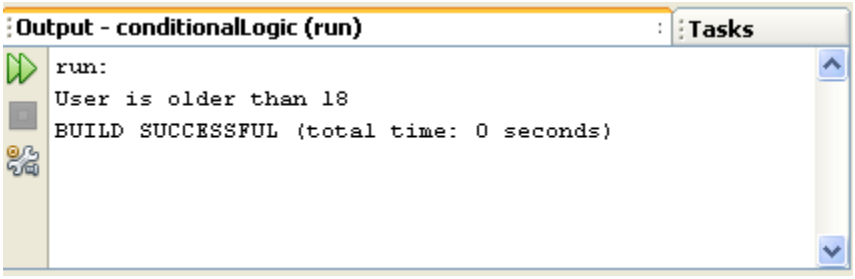
You can have more than one IF Statement in your code. Try the following code:

```
public static void main(String[] args) {

    int user = 18;

    if (user <= 18) {
        System.out.println("User is 18 or younger");
    }

    if (user > 18) {
        System.out.println("User is older than 18");
    }

}
```

This time, we have two IF Statements. The first tests for values less than or equal to 18. The second tests for values greater than 18. When the code is run with a value of 18 or less for the user variable, the Output is this:



Changing the value of the user variable to 20 gives this:



So only one of the IF Statements will Output a print line. And it all depends on what the value of the user variable is.

IF … ELSE

Instead of using two IF Statements, you can use an IF … ELSE Statement instead. Here's the structure of an IF … ELSE statement:

if (condition_to_test **) {**

}
else {

}

The first line starts with if, followed by the condition you want to test for. This goes between two round brackets. Again, curly brackets are used to section off the different choices. The second choice goes after the word else and between its own curly brackets. Here's our code again that checks a user's age:

```
public static void main(String[] args) {

    int user = 17;

    if (user <= 18) {
        System.out.println("User is 18 or younger");
    }
    else {
        System.out.println("User is older than 18");
    }
}
```

So there are only two choices here: either the user is 18 or younger, or the user is older than that. Adapt your code to match that in the image above and try it out. You should find that the first message prints out. Now change the value of the user variable to 20 and run the code again. The message between the ELSE curly brackets should display in the Output window.

IF ... ELSE IF

You can test for more than two choices. For example, what if we wanted to test for more age ranges, say 19 to 39, and 40 and over? For more than two choices, the IF … ELSE IF statement can be used. The structure of an IF … ELSE IF is this:

```
if ( condition_one ) {

}
else if ( condition_two ) {

}
else {

}
```

The new part is this:

```
else if ( condition_two ) {

}
```

So the first IF tests for condition number one (18 or under, for example). Next comes else if, followed by a pair of round brackets. The second condition goes between these new round brackets. Anything not caught by the first two conditions will be caught be the final else. Again, code is sectioned off using curly brackets, with each if, else if, or else having its own pair of curly brackets. Miss one out and you'll get error messages.

Before trying out some new code, you'll need to learn some more conditional operators. The ones you have used so far are these:

> Greater Than
< Less Than
>= Greater Than or Equal To
<= Less Than or Equal To

Here's four more you can use:

&& AND
|| OR
== HAS A VALUE OF
! NOT

The first one is two ampersand symbols, and is used to test for more than one condition at the same time. We can use it to test for two age ranges:

else if (user > 18 && user < 40)

Here, we want to check if the user is older than 18 but younger than 40. Remember, we're trying to check what is inside of the user variable. The first condition is "Greater than 18" (user > 18). The second condition is "Less than 40" (user < 40). In between the two we have our AND operator (&&). So the whole line says "else if user is greater than 18 AND user is less than 40."

We'll get to the other three conditional operators in a moment. But here's some new code to try out:

```
public static void main(String[] args) {

    int user = 21;

    if (user <= 18) {
        System.out.println("User is 18 or younger");
    }
    else if (user > 18 && user < 40) {
        System.out.println("User is between 19 and 39");
    }

    else {
        System.out.println("User is older than 40");
    }
}
```

Run your programme and test it out. You should be able to guess what it will print out before running it. Because we have a value of 21 for the user variable the message between the curly brackets of else if will display in the Output window.

Exercise
Change the value of the user variable from 21 to 45. The message for the else section of the code should now display.

You can add as many else if parts as you want. Suppose we wanted to check if the user was either 45 or 50. We can use two of the new conditional operators above. We can check if the user variable "has a value of 45" OR "has a value of 50":

else if (user == 45 || user == 50)

To test if the user variable has a value of something you use two equal signs, with no space between them. Java will test for that value and no other values. Because want to test for the user

being 50 as well, we can have another condition in the same round brackets: user == 50. This just says "test if the user variable has a value of 50". In between the two conditions, we have the OR operator. This is two pipe characters, which is just to the left of the letter "z" on a UK keyboard. Again, there's no space between the two. The whole of the line above says "Else if the user has a value of 45 OR the user has a value of 50".

Here's our code with the new else if part:

```
public static void main(String[] args) {

    int user = 45;

    if (user <= 18) {
        System.out.println("User is 18 or younger");
    }
    else if (user > 18 && user < 40) {
        System.out.println("User is between 19 and 39");
    }
    else if (user == 45 || user == 50) {
        System.out.println("User is either 45 OR 50");
    }
    else {
        System.out.println("User is older than 40");
    }
}
```

Try it out for yourself. Change the value of the user variable to 45 and run your code. Then change it to 50 and run the code again. In both cases the new message should display.

The various conditional operators can be tricky to use. But you're just testing a variable for a particular condition. It's simply a question of picking the right conditional operator or operators for the job.

Nested IF Statements

You can nest IF Statements. (This also applies to IF ... ELSE and IF ... ELSE IF statements.) Nesting an IF Statement just means putting one IF Statement inside of another. For example, suppose you want to find out if somebody is younger than 18, but older than 16. You want to display a different message for the over 16s. You start with the first IF Statement:

```
if ( user < 19 ) {
System.out.println( "18 or younger");
}
```

To check for over 16, you can place a second IF Statement inside of the one you already have. The format is the same:

```
if ( user < 19 ) {
if ( user > 16 && user < 19 ) {
System.out.println( "You are 17 or 18");
}
}
```

So the first IF Statement catches the user variable if it's less than 19. The second IF Statement narrows the user variable down even further, for ages over 16 and under 19. To print different messages, you can have an IF ... ELSE statement instead of the IF Statement above:

```
if ( user < 19 ) {

if ( user > 16 && user < 19 ) {
System.out.println( "You are 17 or 18");
}
else {
System.out.println( "16 or younger");
}

}
```

Notice where all the curly brackets are in the code: get one wrong and your programme won't run.

Nested IF Statements can be tricky, but all you're trying to do is to narrow down the choices.

Boolean Values

A Boolean value is one with two choices: true or false, yes or no, 1 or 0. In Java, there is a variable type for Boolean values:

```
boolean user = true;
```

So instead of typing int or double or string, you just type boolean (with a lower case "b"). After the name of you variable, you can assign a value of either true or false. Notice that the assignment operator is a single equals sign (=). If you want to check if a variable "has a value of" something, you need two equal signs (= =).

Try this simple code:

```
boolean user = true;
```

```
if ( user == true) {
System.out.println("it's true");
}
else {
System.out.println("it's false");
}
```

So the first IF Statement checks if the user variable has a value of true. The else part checks if it is false. You don't need to say "else if (user = = false)". After all, if something is not true then it's false. So you can just use else: there's only two choices with boolean values.

The only other conditional operator on our lists is the NOT operator. You can use this with boolean values. Have a look at the following code:

```
boolean user = true;

if ( !user ) {
System.out.println("it's flase");
}
else {
System.out.println("it's true");
}
```

It's almost the same as the other boolean code, except for this line:

```
if ( !user ) {
```

This time, we have our NOT operator before the user variable. The NOT operator is a single exclamation mark (!) and it goes before the variable you're tying to test. It's testing for negation, which means that it's testing for the opposite of what the value actually is. Because the user variable is set to true then !user will test for false values. If user was set to false then !user would test for true values. Think of it like this: if something is NOT true then what is it? Or if it's NOT false then what?

Switch Statements in Java

Another way to control the flow of your programmes is with something called a switch statement. A switch statement gives you the option to test for a range of values for your variables. They can be used instead of long, complex **if … else if** statements. The structure of the switch statement is this:

```
switch ( variable_to_test ) {
case value:
code_here;
break;
```

```
case value:
code_here;
break;
default:
values_not_caught_above;

}
```

So you start with the word **switch**, followed by a pair of round brackets. The variable you want to check goes between the round brackets of switch. You then have a pair of curly brackets. The other parts of the switch statement all go between the two curly brackets. For every value that you want to check, you need the word **case**. You then have the value you want to check for:

case value:

After case value comes a colon. You then put what you want to happen if the value matches. This is your code that you want executed. The keyword **break** is needed to break out of each case of the switch statement.

The default value at the end is optional. It can be included if there are other values that can be held in your variable but that you haven't checked for elsewhere in the switch statement.

If all of that is confusing, here's some code to try out. You can either start a new project for this, or just comment out the code you have. A quick way to comment out code in NetBeans is from the toolbar at the top. First, highlight the code you want to comment out. Then click the comment icon:

But here's the code:

```
public static void main(String[] args) {

    int user = 18;

    switch ( user ) {
        case 18:
            System.out.println("You're 18");
            break;
      case 19:
            System.out.println("You're 19");
            break;
      case 20:
            System.out.println("You're 20");
            break;
      default:
          System.out.println("You're not 18, 19 or 20");
    }

}
```

The first thing the code does is to set a value to test for. Again, we've set up an integer variable and called it **user**. We've set the value to 18. The switch statement will check the user variable and see what's in it. It will then go through each of the case statements in turn. When it finds one that matches, it will stop and execute the code for that case. It will then break out of the switch statement.

Try the programme out. Enter various values for the user variable and see what happens.

Sadly, you can't test for a range of values after case, just the one value. So you can't do this:

case (user <= 18):

But you can do this:

case 1: case 2: case 3: case 4:

So the above line tests for a range of values, from 1 to 4. But you have to "spell out" each value. (Notice where all the case and colons are.)

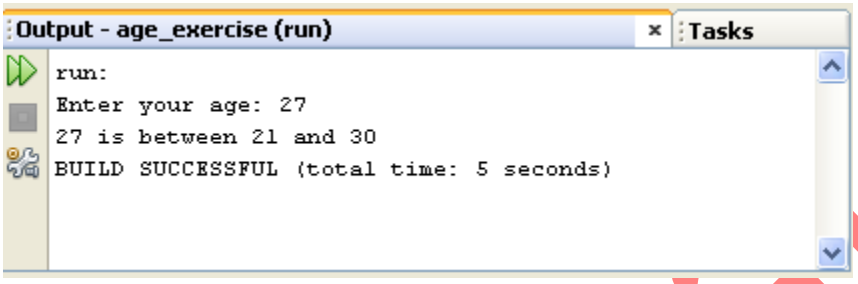
To end this section on conditional logic, try these exercises.

Exercise
Write a programme that accepts user input from the console. The programme should take a

number and then test for the following age ranges: 0 to 10, 11 to 20, 21 to 30, 30 and over. Display a message in the Output window in the following format:

user_age + " is between 21 and 30"

So if the user enters 27 as the age, the Output window should be this:



If the user is 30 or over, you can just display the following message:

"Your are 30 or over"

Help for this exercise

To get string values from the user, you did this:

String age = user_input.next( );

But the **next**( ) method is used for strings. The age you are getting from the user has to be an integer, so you can't use next( ). There is, however, a similar method you can use: **nextInt**( ).

Exercise
If you want to check if one String is the same as another, you can use a Method called **equals**.

```
String user_name = "Bill";

if ( user_name.equals( "Bill" ) ) {
//DO SOMETHING HERE
}
```

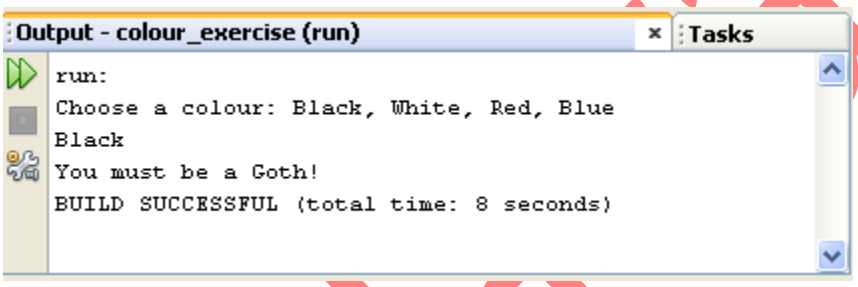
In the code above, we've set up a String variable and called it user_name. We've then assigned a value of "Bill" to it. In between the round brackets of IF we have the variable name again, followed by a dot. After the dot comes the word "equals". In between another pair of round brackets you type the string you're trying to test for.

NOTE: When checking if one string is the same as another, they have to match exactly. So "Bill" is different from "bill". The first has an uppercase letter "B" and the second has a lowercase "b".

For this exercise, write a programme that asks a user to choose between four colours: black, white, red, or blue. Use IF … ELSE IF statements to display one of the following messages, depending on which colour was chosen:

BLACK "You must be a Goth!"
WHITE "You are a very pure person"
RED "You are fun and outgoing"
BLUE "You're not a Chelsea fan, are you?"

When your programme ends, the Output window should look like something like this:



Loops in Java

As we mentioned earlier, the programming you are doing now is sequential programming. This means that flow is downward, from top to bottom, with every line of code being executed, unless you tell Java otherwise.

You saw in the last section that one way to "tell" Java not to execute every line is by using IF Statement to section off areas of code.

Another way to interrupt the flow from top to bottom is by using loops. A programming loop is one that forces the programme to go back up again. If it is forced back up again you can execute lines of code repeatedly.

As an example, suppose you wanted to add up the numbers 1 to 10. You could do it quite easily in Java like this:

int addition = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10;

But you wouldn't really want to use that method if you needed to add up the numbers 1 to a 1000. Instead, you can use a loop to go over a line of code repeatedly until you've reached 1000. Then you can exit the loop and continue on your way.

Java For Loops

We'll start with For Loops, one of the most common types of loops. The "For" part of "For Loop" seems to have lost its meaning. But you can think of it like this: "Loop **FOR** a set number of times." The structure of the For Loop is this:

```
for ( start_value; end_value; increment_number ) {

//YOUR_CODE_HERE

}
```

So after the word "for" (in lowercase) you have a pair of round brackets. Inside of the round brackets you need three things: the start value for the loop, the end value for the loop, and a way to get from one number to another. This is called the increment number, and is usually 1. But it doesn't have to be. You can go up in chunks of 10, if you want.

After the round brackets you need a pair of curly brackets. The curly brackets are used to section off the code that you want to execute repeatedly. An example might clear things up.

Start a new project for this. Call the Project and Class anything you like. (We've called our Project "loops", and the Class "ForLoops"). Now add the following code:

```
package loops;


public class ForLoops {

    public static void main(String[] args) {

        int loopVal;
        int end_value = 11;

        for (loopVal = 0; loopVal < end_value; loopVal++) {

            System.out.println("Loop Value = " + loopVal);
        }
    }

}
```

We start by setting up an integer variable, which we've called loopVal. The next line sets up another integer variable. This variable will be used for the end value of the loop, and is set to 11. What we're going to do is to loop round printing out the numbers from 0 to 10.

Inside the round brackets of the for loop, we have this:

loopVal =0; loopVal < end_value; loopVal++

The first part tells Java at what number you want to start looping. Here, we're assigning a value of zero to the loopVal variable. This will be the first number in the loop. The second part uses some conditional logic:

loopVal < end_value

This says "loopVal is less than end_value". The for loop will then keep going round and round while the value inside the loopVal variable is less than the value in the variable called end_value. As long as it's true that loopVal is less than end_value, Java will keep looping over the code between the curly brackets.

The final part between the round brackets of the for loop is this:

loopVal++

What we're doing here is telling Java how to go from the starting value in loopVal to the next number in the sequence. We want to count from 0 to 10. The next number after 0 is 1. loopVal++ is a shorthand way of saying "add 1 to the value in the variable".

Instead of saying loopVal++ we could have wrote this:

loopVal = loopVal + 1

To the right of the equals sign we have loopVal + 1. Java will then add 1 to whatever is currently stored in the loopVal variable. Once it has added 1 to the value, it will store the result inside of the variable to the left of the equals sign. This is the loopVal variable again. The result is that 1 keeps getting added to loopVal. This is called incrementing the variable. It is so common that the shorthand notation variable++ was invented:

```
int some_number = 0;
some_number++;
```

The value of some_number will be 1 when the code above is executed. It is the short way of saying this:

```
int some_number = 0;
some_number = some_number + 1;
```

To recap then, our for loop is saying this:

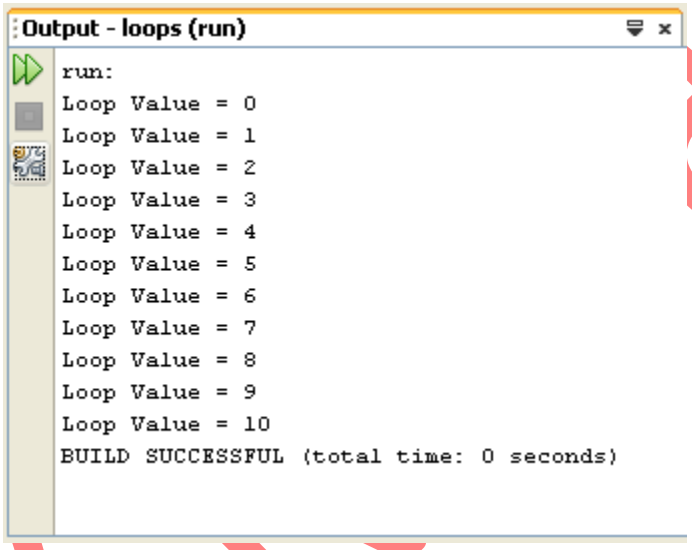
Loop Start value: 0
Keep Looping While: Start value is less than 11
How to advance to the end value: Keep adding 1 to the start value

Inside of the curly brackets of the for loop we have this:

System.out.println("Loop Value = " + loopVal);

Whatever is currently inside of the loopVal variable will be printed out, along with some text.

Run your programme and you should see this in the Output window:



So we've trapped the programme in a loop, and forced it to go round and round. Each time round the loop, 1 gets added to the **loopVal** variable. The loop keeps going round and round while the value inside of **loopVal** is less than the value in **end_value**. Whatever is inside of the loop's curly brackets is the code that will be executed over and over. And that is the whole point of the loop: to execute the curly bracket code over and over.

Here's some code that adds up the numbers 1 to 10. Try it out:

```
public static void main(String[] args) {

    int loopVal;
    int end_value = 11;
    int addition = 0;

    for (loopVal = 1; loopVal < end_value; loopVal++) {

        addition = addition + loopVal;
    }

    System.out.println("Total = " + addition);
}
```

The answer you should get in the Output window is 55. The code itself is more or less the same as the previous for loop. We have the same two variables set up at the top, **loopVal** and **end_value**. We also have a third integer variable, which we've called **addition**. This will hold the value of the 1 to 10 sum.

In between the round brackets of the for loop, it's almost the same as last time: we're looping while **loopVal** is less than **end_value**; and we're adding 1 to the loopVal variable each time round the loop (loopVal++). The only difference is that the starting value is now 1 (loopVal=1).

In between the curly brackets, we only have one line of code:

addition = addition + loopVal;

This single line of code adds up the numbers 1 to 10. If you're confused as to how it works, start to the right of the equals sign:

addition + loopVal;

The first time round the loop, the **addition** variable is holding a value of 0. The variable **loopVal**, meanwhile, is holding a value of 1 (its starting value). Java will add 0 to 1. Then it will store the result to the variable on the left of the equals sign. Again, this is the addition variable. Whatever was previously being held in the addition variable (0) will be erased, and replaced with the new value (1).

The second time round the loop, the values in the two variables are these (the values are between round brackets):

addition (1) + loopVal (2);

1 + 2 is obviously 3. So this is the new value that will be stored to the left of the equals sign.

The third time round the loop, the new values are these:

addition (3) + loopVal (3);

Java adds the 3 + 3 and stores 6 to the left of the equals sign. It keeps going round and round until the loop ends. The result is 55.

(Notice that our print line is outside of the for loop, after the final curly bracket of the loop.)

Exercise
Change you code so that the loop adds up the numbers 1 to a 100. The answer you should get is 5050.

Exercise
Write a times table programme. The programme should ask a user to input a number. This number is then used as the times table. So if the user enters 10, the 10 times table should be displayed. Your Output window should look something like this, when your programme is run.

```
run:
Which times table do you want?
 10
1 times 10 = 10
2 times 10 = 20
3 times 10 = 30
4 times 10 = 40
5 times 10 = 50
6 times 10 = 60
7 times 10 = 70
8 times 10 = 80
9 times 10 = 90
10 times 10 = 100
BUILD SUCCESSFUL (total time: 3 seconds)
```

Help for this Exercise

Your **for** loop only needs two lines of code between the curly brackets, and one of those is the print line. You only need a single line to calculate the answers for your times table.

You should already know how to get the number from the user. This can be used in your loop's curly brackets to work out the answer to your multiplication.

Your times table only needs to go from 1 to 10, like ours does in the image above.

Exercise

Use a **for** loop to print out the odd numbers from 1 to 10. (For the easy way to do this exercise, think about the increment value of the loop, which is the third item between the round brackets.)

One of the hard ways to do the exercise above is by using an operator you haven't yet met - the modulus operator. Modulus is when you divide by a number and keep the remainder. So 10 Mod 3 is 1, because 10 divide by 3 is 3. The remainder is 1, and that's what you keep. The Modulus operator in Java is the percentage sign, rather confusingly. It's used like this:

```
int remainder;
int total = 10
remainder = total %3
```

So the number (or variable) you want to divide up goes first. You then type a percentage sign, followed by your divider number. The answer is the remainder.

In the exercise above, you could use 2 as the Mod number, and then use an IF Statement in your for loop to test what the remainder is. (Can you see why 2 should be the Mod number?)

While Loops

Another type of loop you can use in Java is called the **while** loop. While loops are a lot easier to understand than for loops. Here's what they look like:

```
while ( condition ) {

}
```

So you start with the word "while" in lowercase. The condition you want to test for goes between round brackets. A pair of curly brackets comes next, and the code you want to execute goes between the curly brackets. As an example, here's a while loop that prints out some text (Try the code out for yourself):

```
int loopVal = 0;

while ( loopVal < 5) {
System.out.println("Printing Some Text");
loopVal++;
}
```

The condition to test is between the round brackets. We want to keep looping while the variable called **loopVal** is less than 5. Inside of the curly brackets our code first prints out a line of text. Then we need to increment the **loopVal** variable. If we don't we'll have an infinite loop, as there is no way for **loopVal** to get beyond its initial value of 0.

Although we've used a counter (loopVal) to get to the end condition, while loops are best used when you don't really need a counting value, but rather just a checking value. For example, you can keep looping while a key on the keyboard is not pressed. This is common in games programme. The letter "X" can pressed to exit the while loop (called the game loop), and hence the game itself. Another example is looping round a text file while the end of the file has not been reached.

Do ... While

Related to the **while** loop is the **do … while** loop. It looks like this:

```
int loopVal = 0;

do {
System.out.println("Printing Some Text");
loopVal++;
}
while ( loopVal < 5 );
```

Again, Java will loop round and round until the end condition is met. This time, the "while" part is at the bottom. But the condition is the same - keep looping while the value inside of the variable called loopVal is less than 5. The difference between the two is the code between the curly brackets of do … while will get executed at least once. With the while loop, the condition could already be met. Java will then just bail out of your loop, and not even execute your curly bracket code. To test this out, try the while loop first. Change the value of your loopVal variable to 5, and then run the code. You should find that the text doesn't get printed. Now try the do loop with a value of 5 for loopVal. The text will print once and then Java will bail out of the loop.

Java Arrays

A programming concept you just have to get used to if you're to code effectively is the array. In this section, you'll learn what arrays are, and how to use them.

What is an Array?

So far, you have been working with variables that hold only one value. The integer variables you have set up have held only one number, and the string variables just one long string of text. An array is a way to hold more than one value at a time. It's like a list of items. Think of an array as the columns in a spreadsheet. You can have a spreadsheet with only one column, or lots of columns. The data held in a single-list array might look like this:

	Array_Values
0	10
1	14
2	36
3	27
4	43
5	18

Like a spreadsheet, arrays have a position number for each row. The positions in an array start at 0 and go up sequentially. Each position in the array can then hold a value. In the image above array position 0 is holding a value of 10, array position 1 is holding a value of 14, position 2 has a value of 36, and so on.

To set up an array of number like that in the image above, you have to tell Java what kind of data is going in to your array (integers, strings, boolean values, etc). You then need to say how many positions the array has. You set them up like this:

int[ ] aryNums;

The only difference between setting up a normal integer variable and an array is a pair of square brackets after the data type. The square brackets are enough to tell Java that you want to set up an array. The name of the array above is aryNums. Just like normal variables, you can call them almost anything you like (with the same exceptions we mentioned earlier).

But this just tells Java that you want to set up an integer array. It doesn't say how many positions the array should hold. To do that, you have to set up a new array object:

aryNums = new int[6];

You start with your array name, followed by the equals sign. After the equals sign, you need the Java keyword new, and then your data type again. After the data type come a pair of square brackets. In between the square brackets you need the size of the array. The size is how many positions the array should hold.

If you prefer, you can put all that on one line:

int[ ] aryNums = new int[6];

So we are telling Java to set up an array with 6 positions in it. After this line is executed, Java will assign default values for the array. Because we've set up an integer array, the default values for all 6 positions will be zero (0).

To assign values to the various positions in an array, you do it in the normal way:

aryNums[0] = 10;

Here, a value of 10 is being assigned to position 0 in the array called aryNums. Again, the square brackets are used to refer to each position. If you want to assign a value of 14 to array position 1, the code would be this:

aryNums[1] = 14;

And to assign a value of 36 to array position 2, it's this:

aryNums[2] = 36;

Don't forget, because arrays start at 0, the third position in an array has the index number 2.

If you know what values are going to be in the array, you can set them up like this instead:

int[ ] aryNums = { 1, 2, 3, 4 };

This method of setting up an array uses curly brackets after the equals sign. In between the curly brackets, you type out the values that the array will hold. The first value will then be position 0, the second value position 1, and so on. Note that you still need the square brackets after int, but not the new keyword, or the repetition of the data type and square brackets. But this is just for data types of int values, string, and char values. Otherwise, you need the new keyword. So you can do this:

String[ ] aryStrings = {"Autumn", "Spring", "Summer", "Winter" };

But not this:

boolean[ ] aryBools = {false, true, false, true};

To set up a boolean array you still need the new keyword:

boolean[ ] aryBools = new boolean[ ] {false, true, false, true};

To get at the values held in your array, you type the name of the array followed by an array position in square brackets. Like this:

System.out.println(aryNums[2]);

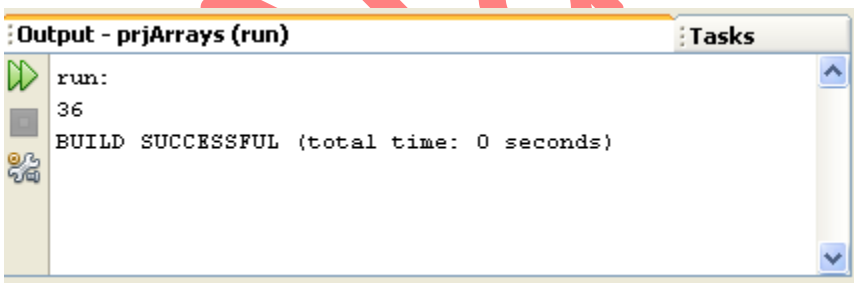
The above code will print out whatever value is held at array position 2 in the array called aryNums. But let's get some coding practice.

Start a new project and call it anything you like. Don't forget to change the name of the Class to something relevant.

Type the following code into your new Main method:

```
package prjarrays;

public class ArraysTest {

    public static void main(String[] args) {

        int[] aryNums;

        aryNums = new int[6];

        aryNums[0] = 10;
        aryNums[1] = 14;
        aryNums[2] = 36;
        aryNums[3] = 27;
        aryNums[4] = 43;
        aryNums[5] = 18;

        System.out.println( aryNums[2] );
    }

}
```

When you run the programme you should see this in the Output window:



Change the array position number in the print line from 2 to 5 and 18 should print out instead.

Arrays and Loops

Arrays come into their own with loops. You have seen in the previous section that to assign values to array positions, you did this:

aryNums[0] = 10;

But that's not terribly practical if you have a lot of numbers to assign to an array. As an example, imagine a lottery programme that has to assign the numbers 1 to 49 to positions in an array. Instead of typing a long list of array positions and values you can use a loop. Here's some code that does just that:

```
package prjarrays;

public class ArraysTest {

    public static void main(String[] args) {

        int[] lottery_numbers = new int[49];
        int i;

        for (i=0; i < lottery_numbers.length; i++) {
            lottery_numbers[i] = i + 1;
            System.out.println( lottery_numbers[i] );
        }

    }
}
```

So we set up an array to hold 49 integer values. Then comes the loop code. Notice the end condition of the loop:

i < lottery_numbers.length

Length is a property of array objects that you can use to get the size of the array (how many positions it has). So this loop will keep going round and round while the value in the variable i is less than the size of the array.

To assign values to each position in the array, we have this line:

lottery_numbers[i] = i + 1;

Instead of a hard-code value between the square brackets of the array name, we have the variable called **i**. This increases by 1 each time round the loop, remember. Each array position can then be accessed just by using the loop value. The value that is being assigned to each position is i + 1. So again, it's just the incremented loop value, this time with 1 added to it. Because the loop value is starting at 0, this will give you the numbers 1 to 49.

The other line in the loop just prints out whatever value is in each array position.
(If you wanted, you could then write code to jumble up the numbers in the array. Once you have jumbled up the values, you could then take the first 6 and use them as the lottery numbers. Write another chunk of code that compares a user's 6 numbers with the winning numbers and you have a lottery programme!)

Sorting Arrays

Other inbuilt java methods allow you to sort your arrays. To use the sort method of arrays, you first need to reference a Java library called **Arrays**. You do this with the import statement. Try it with your aryNums programme. Add the following import statement:

import java.util.Arrays;

You code should look like ours below:

```
package prjarrays;

import java.util.Arrays;

public class ArraysTest {

    public static void main(String[] args) {

        int[] aryNums;
        aryNums = new int[6];

        aryNums[0] = 10;
        aryNums[1] = 14;
        aryNums[2] = 36;
        aryNums[3] = 27;
        aryNums[4] = 43;
        aryNums[5] = 18;

    }

}
```

Now that you have imported the Arrays library, you can use the sort method. It's quite easy:

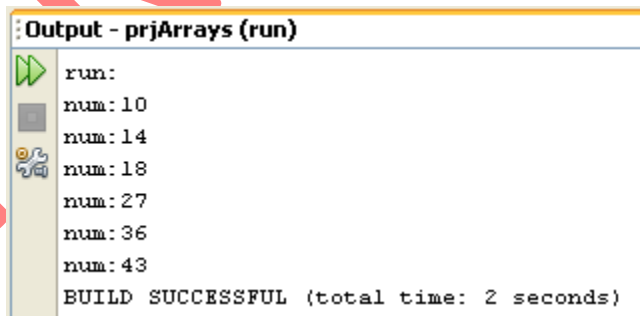
Arrays.sort(aryNums **);**

First you type the word "Arrays", then a dot. As soon as you type a dot, NetBeans will display a list of things you can do with arrays. Type the word "sort". In between a pair of round brackets, you then put the name of the array you want to sort. (Notice that you don't need any square brackets after the array name.)

And that's it - that's enough to sort the array! Here's some code to try out:

```
package prjarrays;

import java.util.Arrays;

public class ArraysTest {

    public static void main(String[] args) {

        int[] aryNums;
        aryNums = new int[6];

        aryNums[0] = 10;
        aryNums[1] = 14;
        aryNums[2] = 36;
        aryNums[3] = 27;
        aryNums[4] = 43;
        aryNums[5] = 18;

        Arrays.sort(aryNums);

        int i;

        for (i=0; i < aryNums.length; i++) {
            System.out.println("num:" + aryNums[i]);
        }

    }
}
```

The for loop at the end will go round and round printing out the values in each array position. When the code is run, the Output will look like this:



As you can see, the array has been sorted in ascending order.

Sorting in descending order, however, is only possible either by writing your own sorting code, or converting your array to Integer objects then importing from the Collections library. If you need to a descending sort, here's some code that does just that (skip this code, if you want):

```
package prjarrays;

import java.util.Arrays;
import java.util.Collections;

public class ArraysTest {

    public static void main(String[] args) {

      int[] aryNums;
      aryNums = new int[6];

      aryNums[0] = 10; aryNums[1] = 14; aryNums[2] = 36;
      aryNums[3] = 27; aryNums[4] = 43; aryNums[5] = 18;

      //CREATE AN INTEGER OBJECT ARRAY
      Integer[] integerArray = new Integer[aryNums.length];

      //ASSIGN THE VALUES TO THE NEW ARRAY
      for (int i = 0; i < aryNums.length; i++) {
          integerArray[i] = new Integer(aryNums[i]);
      }

      //SORT DESCENDING
      Arrays.sort(integerArray,  Collections.reverseOrder() );

      //PRINT THE RESULTS
      for (int i =0; i < integerArray.length; i++) {
          System.out.println("num:" + integerArray[i]);
      }
    }
}
```

All a bit messy, I'm sure you'll agree!

Arrays and Strings

You can place strings of text into arrays. This is done in the same way as for integers:

String[ ] aryString = new String[5] ;

aryString[0] = "This";
aryString[1] = "is";
aryString[2] = "a";
aryString[3] = "string";
aryString[4] = "array";

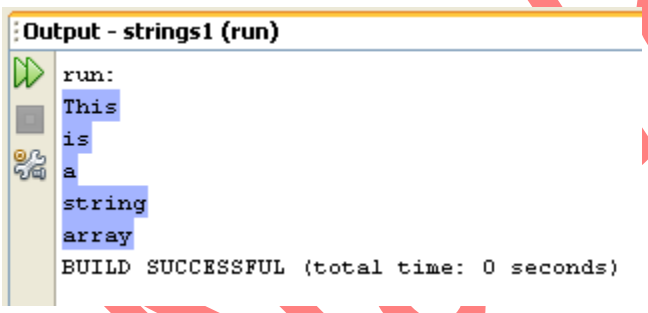
The code above sets up a string array with 5 positions. Text is then assigned to each position in the array.

Here's a loop that goes round all the positions in the array, printing out whatever is at each position:

```
int i;
for ( i=0; i < aryString.length; i++ ) {
System.out.println( aryString[i] );
}
```

The loop goes round and round while the value in the variable called i is less than the length of the array called aryString.

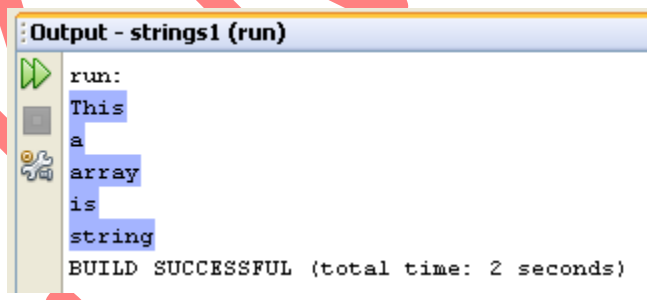
When the above programme is run, the Output window will look like this:



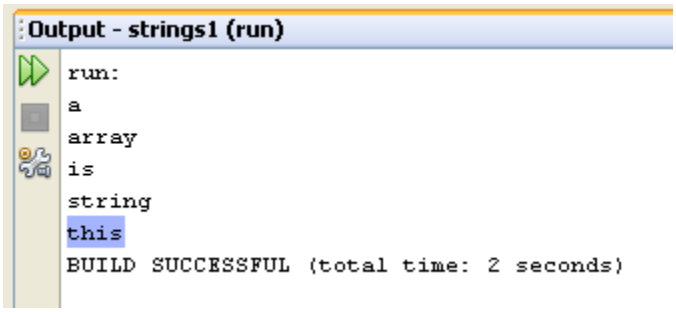
You can perform a sort on string arrays, just like you can with integers. But the sort is an alphabetical ascending one, meaning that "aa" will come first over "ab". However, Java uses Unicode characters to compare one letter in your string to another. This means that uppercase letter will come before lowercase ones. Try the following code:

```
package strings1;
import java.util.Arrays;

public class StringArrays {

    public static void main(String[] args) {

        String[] aryString = new String[5] ;

        aryString[0] = "This";
        aryString[1] = "is";
        aryString[2] = "a";
        aryString[3] = "string";
        aryString[4] = "array";

        Arrays.sort(aryString);

        int i;
        for (i=0; i < aryString.length; i++){
            System.out.println( aryString[i] );
        }

    }
}
```

When the programme is run, the Output window will display the following:



Although we've sorted the array, the word "This" comes first. If this were an alphabetical sort, you'd expect the word "a" to come first." And it does if all the letters are lowercase. In your programming code, change the capital "T" of "This" to a lowercase "t". Now run your programme again. The Output window will now display the following:



As you can see, the word "this" is now at the bottom. We'll have a closer look at strings in the next section, so don't worry too much about them now. Instead, try these exercises.

Exercise
Set up an array to hold the following values, and in this order: 23, 6, 47, 35, 2, 14. Write a programme to get the average of all 6 numbers. (You can use integers for this exercise, which will round down your answer.)

Exercise
Using the above values, have your programme print out the highest number in the array.

Exercise
Using the same array above, have your programme print out only the odd numbers.

Multi-Dimensional Arrays in Java

The arrays you have been using so far have only held one column of data. But you can set up an array to hold more than one column. These are called multi-dimensional arrays. As an example, think of a spreadsheet with rows and columns. If you have 6 rows and 5 columns then your spreadsheet can hold 30 numbers. It might look like this:

	A	B	C	D	E
0	10	12	43	11	22
1	20	45	56	1	33
2	30	67	32	14	44
3	40	12	87	14	55
4	50	86	66	13	66
5	60	53	44	12	11

A multi dimensional array is one that can hold all the values above. You set them up like this:

int[ ][ ] aryNumbers = new int[6**][**5**];**

They are set up in the same way as a normal array, except you have two sets of square brackets. The first set of square brackets is for the rows and the second set of square brackets is for the columns. In the above line of code, we're telling Java to set up an array with 6 rows and 5 columns. To hold values in a multi-dimensional array you have to take care to track the rows and columns. Here's some code to fill the first rows of numbers from our spreadsheet image:

aryNumbers[0][0] = 10;
aryNumbers[0][1] = 12;
aryNumbers[0][2] = 43;
aryNumbers[0][3] = 11;
aryNumbers[0][4] = 22;

So the first row is row 0. The columns then go from 0 to 4, which is 5 items. To fill the second row, it would be this:

aryNumbers[1][0] = 20;
aryNumbers[1][1] = 45;
aryNumbers[1][2] = 56;
aryNumbers[1][3] = 1;
aryNumbers[1][4] = 33;

The column numbers are the same, but the row numbers are now all 1.

To access all the items in a multi-dimensional array the technique is to use one loop inside of another. Here's some code to access all our number from above. It uses a double for loop:

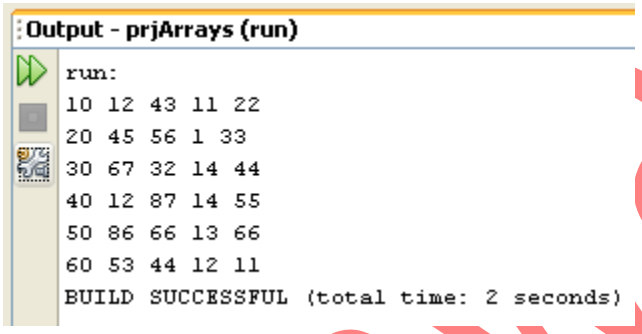
```
int rows = 6;
int columns = 5;

int i, j;

for (i=0; i < rows ; i++) {

    for (j=0; j < columns ; j++) {
        System.out.print( aryNumbers[i][j] + " " );
    }
    System.out.println( "" );
}
```

The first for loop is used for the rows; the second for loop is for the columns. The first time round the first loop, the value of the variable i will be 0. The code inside of the for loop is another loop. The whole of this second loop will be executed while the value of the variable i is 0. The second for loop use a variable called j. The i and the j variables can then be used to access the array.

aryNumbers[i][j]

So the two loop system is used to go through all the values in a multi-dimensional array, row by row.

Exercise
Finish off the programme above where we are writing a programme to print out all the values from the spreadsheet. Your Output window should look something like this when you're done:



Multi-dimensional arrays can be quite tricky, but mainly because it's hard to keep track of all your rows and columns!

Array Lists in Java

If you don't know how many items are going to be held in your array, you may be better off using something called an **ArrayList**. An ArrayList is a dynamic data structure, meaning items can be added and removed from the list. A normal array in java is a static data structure, because you stuck with the initial size of your array.

To set up an ArrayList, you first have to import the package from the **java.util library**:

import java.util.ArrayList;

You can then create a new ArrayList object:

ArrayList listTest = new ArrayList( );

Notice that you don't need any square brackets this time.

Once you have a new ArrayList objects, you can add elements to it with the add method:

listTest.add("first item");
listTest.add("second item");

```
listTest.add( "third item" );
listTest.add( 7 );
```

In between the round brackets of add you put what it is you want to add to the ArrayList. You can only add objects, however. The first three items we've added to the list above are String objects. The fourth item is a number. But this will be a number object of type Integer, rather than the primitive data type int.

Items in the list can be referenced by an Index number, and by using the get method:

```
listTest.get( 3 )
```

This line will get the item at Index position 3 on the list. Index numbers start counting at zero, so this will be the fourth item.

You can also remove items from an ArrayList. You can either use the Index number:

```
listTest.remove(2);
```

Or you can use the value on the list:

```
listTest.remove( "second item" );
```

Removing an item will resize the ArrayList, so you have to be careful when trying to get an item on the list when using its Index number. If we've removed item number 2, then our list above will contain only 3 items. Trying to get the item with the Index number 3 would then result in an error.

To go through each item in your ArrayList you can set up something called an **Iterator**. This class can also be found in the **java.util** library:

```
import java.util.Iterator;
```

You can then attach your ArrayList to a new Iterator object:

```
Iterator it = listTest.iterator( );
```

This sets up a new Iterator object called it that can be used to go through the items in the ArrayList called listTest. The reason for using an Iterator object is because it has methods called next and hasNext. You can use these in a loop:

```
while ( it.hasNext( ) ) {
System.out.println( it.next( ) );
}
```

The method hasNext returns a Boolean value. The value will be false if there are no more items in the ArrayList. The next method can be used to go through all the items in the list.

To test all this theory out, try the following code:

```
public static void main(String[] args) {

    ArrayList listTest = new ArrayList();

    listTest.add("first item");
    listTest.add("second item");
    listTest.add("third item");
    listTest.add(7);

    Iterator it = listTest.iterator();

    while (it.hasNext()) {
        System.out.println(it.next());
    }

    //  REMOVE AN ITEM FROM THE LIST
    listTest.remove("second item");

    //PRINT OUT THE NEW LIST
    System.out.println("Whole list=" + listTest);

    //GET THE ITEM AT INDEX POSITION 1
    System.out.println("Position 1=" + listTest.get(1));
}
```

Notice the line that prints out the entire list:

System.out.println("Whole list=" + listTest);

This gives you a quick way to see which items are on your list, if it gets a bit too long.

When the code is run, the Output window will display the following:

first item
second item
third item
7
Whole list=[first item, third item, 7]
Position 1=third item

To sum up, then, use an ArrayList when you're not sure how many elements are going to be in a list of items.

More on Java Strings

There's more to strings than meets the eye. Unlike int variables, or double variables, strings are objects. What this means in practice is that you can do things with strings of text that you can't do with int or double variables. (The same applies to the primitive data types boolean, byte, single, char, float, long and short: they are not objects like strings are.)

Before we get to manipulating strings of text, here's some basic information on what strings actually are.

How Java Stores Strings

A string is a series of Unicode characters held under a variable name. Take the following string:

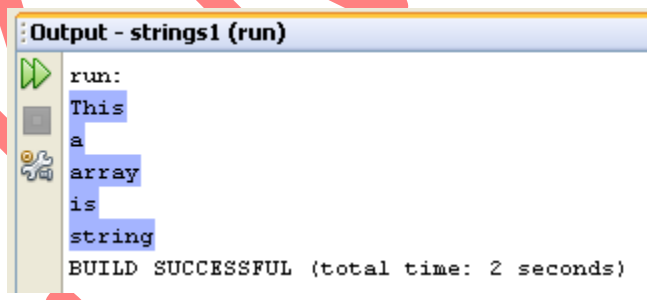
String someText = "Bill";

This tells Java to set up a string object with the four characters "B", "i", "l" and another "l". In the Unicode character set, these values are: \u0042, \u0069, \u006c, \u006c. Unicode values are stored as hexadecimals numbers. Capital letters (A to Z) are stored using the values \u0041 to \u005a, while lowercase letters (a to z) are stored using the hexadecimals values \u0061 to \u007a.

In the previous section, we had an array which held strings of text. We then sorted the array:

```
package strings1;
import java.util.Arrays;

public class StringArrays {

    public static void main(String[] args) {

        String[] aryString = new String[5] ;

        aryString[0] = "This";
        aryString[1] = "is";
        aryString[2] = "a";
        aryString[3] = "string";
        aryString[4] = "array";

        Arrays.sort(aryString);

        int i;
        for (i=0; i < aryString.length; i++){
            System.out.println( aryString[i] );
        }

    }
}
```

When the programme is run, the output is this:



We noted that the word "This" comes first. If the array is supposed to be sorted alphabetically, however, you would expect the word "a" to come first. The reason it doesn't is because lowercase "a" has a hexadecimal value of u\0061, which is the decimal number 97. But uppercase "T" has a hexadecimal value of u\0054, which is the decimal number 84. 84 is lower than 97, so the "T" comes first.

OK, let's do some work manipulating strings of text. The string methods we'll take a look at are:

toUpperCasse
toLowerCase
compareTo
IndexOf
endWith, startsWith
Substring
Equals
charAt
trim
valueOf

Converting to Upper and Lower Case in Java

Converting your Java strings of text to upper or lower case is fairly straightforward: just use the inbuilt methods toUpperCase and toLowerCase.

Start a new project for this, and add the following code:

```
package prjstrings;

public class StringManipulation {

    public static void main(String[] args) {

        String changeCase = "text to change";
        System.out.println( changeCase );

        String result;
        result = changeCase.toUpperCase();

        System.out.println( result );
    }
}
```

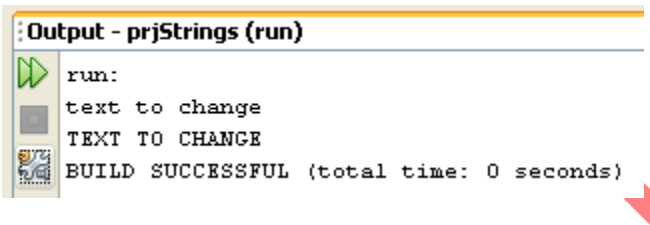
The first two lines of code just set up a String variable to hold the text "text to change", and then we print it out. The third line sets of a second String variable called result. The fourth line is where we do the converting:

result = changeCase.toUpperCase( );

To use a string method you first type the string you want to work on. For us, this was the string in the variable called **changeCase**. Type a dot after the variable name and you'll see a list of available methods that you can use on your string. Select **toUpperCase**. (The method needs the empty round brackets after it.)

After Java has changed the word to uppercase letters, we're storing the new string into our result variable.

When the programme is run, the Output window displays the following:



But you don't have to store the converted word in a new variable. This would work just as well:

System.out.println(changeCase.toUpperCase( ));

Here, Java will just get on with converting the string, without needing to assign the result to a new variable.

If you want to convert to lowercase, just use the **toLowerCase** method instead. It is used in exactly the same way as **toUpperCase**.

Comparing Strings

You can compare one string to another. (When comparing, Java will use the hexadecimal values rather than the letters themselves.) For example, if you wanted to compare the word "Ape" with the word "App" to see which should come first, you can use an inbuilt string method called **compareTo**. Let's see how it works.

You don't need to start a new project for this: simply comment out (or delete) the code you already have. Now add the following code:

```
package prjstrings;

public class StringManipulation {


    public static void main(String[] args) {

        int result;
        String Word1 = "Ape";
        String Word2 = "App";

        result = Word1.compareTo(Word2);

        if (result < 0) {
            System.out.println("Word1 is less than Word2");
        }
        else if (result > 0) {
            System.out.println("Word1 is more than Word2");
        }
        else if (result == 0) {
            System.out.println("The same word");
        }
    }

}
```

We've set up two string variables to contain the words "Ape" and "App". The compareTo method is then this line in the code above:

result = Word1.compareTo(Word2);

The **compareTo** method returns a value. The value that is returned will be greater than 0, less than 0, or have a value of zero. If Word1 comes before Word2, then the value that is returned will be less than 0. If Word1 comes after Word2 then the value returned will be greater than 0. If the two words are identical then a value of 0 will be returned.

So you need to assign the value that compareTo returns to a variable. We're placing the value in an integer variable called result. The IF Statements in the code simply tests to see what is in the result variable

However, when you compare one string of text with another, Java compares the underlying hexadecimals values, rather than the actual letters. Because uppercase letters have a lower hexadecimal value than lowercase ones, an uppercase letter "A" in "App" will come before a lowercase letter "a" in "ape". Try it for yourself. Change "Ape" to "ape" in your code. The Output will read "Word1 is more than Word2", meaning that Java will place the word "ape" after the word "app" alphabetically.

To solve the problem, there's a related method called compareToIgnoreCase. As its name suggest, lowercase and uppercase letter are ignored. Use this and "ape will come before "App" alphabetically.

The Java Method indexOf

The **indexOf** method is used to locate a character or string within another string. For example, you can use it to see if there is a @ character in an email address. Let's use that example in some code.

Again, you can either delete or comment out the code you already have. But here's the new code to try:

```
package prjstrings;

public class StringManipulation {

    public static void main(String[] args) {

        char ampersand = '@';
        String email_address = "meme@me.com";

        int result;
        result = email_address.indexOf( ampersand );

        System.out.println( result );
    }
}
```

We want to check if the @ sign is in the email address, so we first set up a char variable and assign it a value of '@'. (Note the single quotes for the char variable). After setting up an email address, we have a result variable. This is an int variable. The reason that result is an integer is because the **indexOf** method will return a value. It will return the position number of the ampersand character in the string email_address. Here's the relevant line:

result = email_address.indexOf(ampersand);

The string you're trying to search comes first. After a dot, type **indexOf**. In between the round brackets of indexOf, you have several options. One of the options is to type a single character (or the name of char variable). We've placed our ampersand variable between the round brackets of indexOf. Java will then tell us the position of the @ character in the email address. It will store the value in the result variable.

When you run the code, the output will be 4. You might think that the @ sign is the fifth character in the email address. But indexOf starts counting at 0.

However, if the character is not in the word that you're searching, indexOf will return a value of -1. To test this out, delete the @ sign from your email_address string. Then run your code again. You'll see -1 as the output.

You can use the return value of -1 to your advantage. Here's the code again, only with an IF statement that examines the value of the result variable:

```
package prjstrings;

public class StringManipulation {

    public static void main(String[] args) {

        char ampersand = '@';
        String email_address = "mememe.com";

        int result;
        result = email_address.indexOf( ampersand );

        if (result== -1 ) {
            System.out.println( "Invalid Email Address" );
        }
        else {
            System.out.println( "Email Address OK" );
        }

    }
}
```

So if the result of indexOf is -1 then we can do one thing, else allow the user to continue.

You can also use indexOf to test for more than one character. The code below checks the email address to see if it ends with ".com":

```
    public static void main(String[] args) {

        String dotCom = ".com";
        String email_address = "meme@me.com";

        int result;
        result = email_address.indexOf( dotCom );

        if (result== -1 ) {
            System.out.println( "Invalid Email Address" );
        }
        else {
            System.out.println( "Email Address OK ");
        }

    }
```

The code is almost identical, except we're now using a String variable to hold the text we want to check for (.com), and not a char variable.

Again, a result of -1 will be returned if the text to search for is not found in the String that comes before the dot of indexOf. Otherwise, indexOf will return the position of the first of the matching character. In the code above, the dot is the seventh character of the email address, when you start counting from 0.

You can also specify a starting position for your searches. In our email address example, we can start searching for the ".com" after the @ symbol. Here's some code that first locates the position of the @ symbol, and then uses that as the start position to search for ".com".

```
public static void main(String[] args) {

    char ampersand = '@';
    String dotCom = ".com";
    String email_address = "meme@me.com";

    int atPos = email_address.indexOf( ampersand );
    int result = email_address.indexOf(dotCom, atPos);

    if (result== -1 ) {
        System.out.println( "Invalid Email Address" );
    }
    else {
        System.out.println( "Email Address OK " + result );
    }
}
```

The new line of code is this one:

result = email_address.indexOf(dotCom, atPos);

The only thing different is the addition of an extra variable between the brackets of indexOf. We still have the string we want to search for (which is whatever text is in the dotcom variable), but we now have a starting position for the search. This is the value of the variable called atPos. We get the atPos value by using indexOf to locate the position of the @ symbol in the email address. Java will then start the search from this position, rather than starting at 0, which is the default.

Ends With ... Starts With

For the programme above, you can also use the inbuilt method endsWith:

Boolean ending = email_address.endsWith(dotcom);

You need to set up a Boolean variable for endsWith, because the method returns an answer of true or false. The string you're trying to test goes between the round brackets of endsWith, and the text you're searching goes before it. If the text is in the search string then a value of true is returned, else it will be false. You can add an if … else statement to check the value:

```
if (ending == false ) {
System.out.println( "Invalid Email Address" );
}
else {
System.out.println( "Email Address OK " );
}
```

The method startsWith is used in a similar way:

Boolean startVal = email_address.startsWith(dotcom);

Again, the return value is a Boolean true or false.

Substring

One really useful method available to you is called substring. This method allows you to grab one chunk of text from another. In our email address programme above, for example, we could grab the last five characters from the address and see if it is co.uk.

To get some practice with substring, we'll write a small Name Swapper game. For this game, we want to change the first two letters of a family name and swap them with the first two letters of a personal name, and vice versa. So if we have this name:

"Bill Gates"

we would swap the "Ga" of "Gates" with the "Bi" of "Bill" to make "Bites". The "Bi" of "Bill" will then be swapped with the "Ga" of "Gates" to make "Gall". The new name printed out would be: "Gall Bites"

We'll use substring for most of this programme. Substring works like this:

String FullName = "Bill Gates";
String FirstNameChars = "";

FirstNameChars = FullName.substring(0, 2);

You set up a string to search, in this case the string "Bill Gates". The string you're trying to search goes after an equals sign. After a dot, type the name of the method, **substring**. There are two ways to use substring, and the difference is in the numbers between the round brackets. We have two numbers in the code above, 0 and 2. This means start grabbing characters at position 0 in the string, and stop grabbing when you have two of them. The two characters are then returned and placed in the variable FirstNameChars. If you always want to go right to the end of the string, you can just do this:

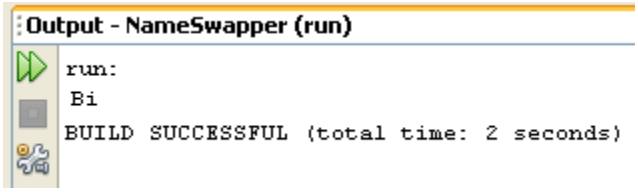
String test = FullName.substring(2);

This time, we only have 1 number between the round brackets of substring. Now, Java will start at character two in the string FirstName, and then grab the characters from position 2 right to the end of the string.

Start a new programme to test this out. Add a print line to the end and your code should be this:

```
package nameswapper;

public class NameSwap {

    public static void main(String[] args) {

        String FullName = "Bill Gates";
        String FirstNameChars = "";

        FirstNameChars = FullName.substring(0, 2);
        System.out.println(FirstNameChars);
    }

}
```

When the programme runs, the Output window should look like this:



So the substring method has allowed us to grab the first two characters of the name "Bill".

To get the first characters, we had a 0 and a 2 between the round brackets of substring. You might think that to get the "Ga" of "Gates" that we could just do this:

= FullName.substring(5, 2);

We still want two characters, after all. Only this time, the 5 would tell Java to start from the "G" of "Gates". (The first position in a string is position 0 not position 1.) So, start at position 5 in the string and grab 2 characters.

However, running that code would get you an error. That's because the second number between the round brackets of substring doesn't mean how many characters you want to grab. It means the position in the string that you want to end at. By specifying 2 we're telling Java to end at the character in position 2 of the string. As you can't go from position 6 backwards to position 2 you get an error instead.

(NOTE: If you start the count at 0 in the string "Bill", you might think that position 2 is the letter "l". And you'd be right. But substring starts before the character at that position, not after it.)

To get the "Ga" of "Gates", therefore, you could do this:

FullName.substring(5, FullName.length( ) - 3);

The second number is now the length of the string minus 3 characters. The length of a string is how many characters it has. "Bill Gates" has 10 characters, including the space. Take away 3 and you have 7. So we're telling substring to start at character 5 and end at character 7.

And that would work perfectly well for people called "Bill Gates". But the programme wouldn't work if you name was, say, "Billy Gates". The code above would then grab the space character plus the letter "G", which is not what we want at all. We want the programme to work whichever two names are entered. So we have to get a bit clever.

One thing we can do is to note the position of the space in the two names. The 2 characters we want to grab from the second name always come right after the space character. We want some code that grabs those first two characters after the space.

We can use indexOf to note the position of the space:

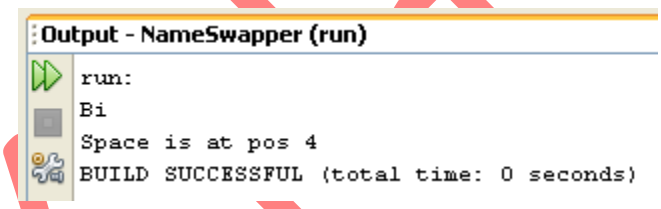
int spacePos = FullName.indexOf(" ");

To specify a space character you can type a space between two double quotes (or single quotes). This then goes between the round brackets of indexOf. The value returned will be an integer, and it is the position of the first occurrence of the space character in the string FullName.

Test it out by adding the line above to your code: Add a print line to check the Output:

```
public static void main(String[] args) {

    String FullName = "Bill Gates";
    String FirstNameChars = "";

    FirstNameChars = FullName.substring(0, 2);
    System.out.println( FirstNameChars );

    int spacePos = FullName.indexOf(" ");
    System.out.println( "Space is at pos "  + spacePos);


}
```

Run the programme to see the following Output:



So the space is at position 4 in the string. We can use this fact to grab the first two characters of "Gates", or indeed any second name. We tell Java to go from the first character after the space, and end at the next two characters:

FullName.substring(spacePos + 1, (spacePos + 1) + 2)

So the two numbers between the round brackets of substring are these:

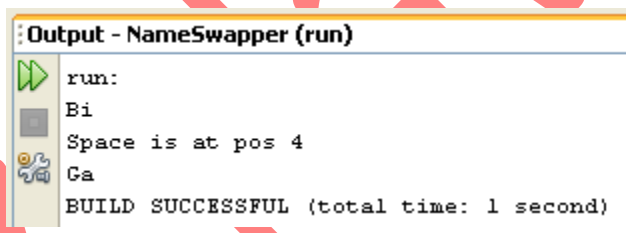
spacePos + 1, (spacePos + 1) + 2

We want to start at the first character after the space (space + 1), and end two characters after this position, which is (spacePos + 1) + 2.

Add the following lines to your code (The ones highlighted. Our new substring method spills over on to two lines, but you can keep your on one, if you prefer):

```
public static void main(String[] args) {

    String FullName = "Bill Gates";
    String FirstNameChars = "";


    FirstNameChars = FullName.substring(0, 2);
    System.out.println( FirstNameChars );

    int spacePos = FullName.indexOf(" ");
    System.out.println( "Space is at pos "  + spacePos);

    String SurNameChars = "";
    SurNameChars = FullName.substring(spacePos + 1,
                                      (spacePos + 1) + 2);

    System.out.println( SurNameChars );

}
```

When you run the programme, the Output window is this:



So we now have the "Bi" from Bill and the "Ga" from Gates. What we now need to do is get the rest of the characters from the two names, and then swap them around.

Again, we can use substring to get the remaining characters from the first name:

String OtherFirstChars = "";
OtherFirstChars = FullName.substring(2, spacePos);
System.out.println(OtherFirstChars);

And the remaining characters from the second name:

String OtherSurNameChars = "";
OtherSurNameChars = FullName.substring((spacePos + 1) + 2,
FullName.length());

System.out.println(OtherSurNameChars);

Not the numbers in between the round brackets of substring. To get the other first name characters, the numbers are these:

2, spacePos

This tells Java to start at position 2, and go right up to the position of the space. To get the rest of the second name, however, it's a little bit trickier:

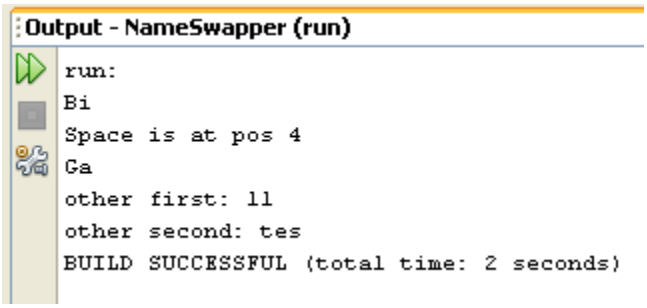
(spacePos + 1) + 2, FullName.length( )

The (spacePos1 + 1) + 2 is the starting position of the third character of the second name. We want to end at the length of the string, which will get us the rest of the characters.

Add the following to your code (highlighted):

```
public static void main(String[] args) {

  String FullName = "Bill Gates";
  String FirstNameChars = "";

  FirstNameChars = FullName.substring(0, 2);
  System.out.println( FirstNameChars );

  int spacePos = FullName.indexOf(" ");
  System.out.println( "Space is at pos "  + spacePos);

  String SurNameChars = "";
  SurNameChars = FullName.substring(spacePos + 1,
                                    (spacePos + 1) + 2);

  System.out.println( SurNameChars );

  String OtherFirstChars = "";
  OtherFirstChars = FullName.substring(2, spacePos);
  System.out.println( "other first: " + OtherFirstChars );

  String OtherSurNameChars = "";
  OtherSurNameChars = FullName.substring((spacePos + 1) + 2,
                                         FullName.length());

  System.out.println("other second: " + OtherSurNameChars );

}
```

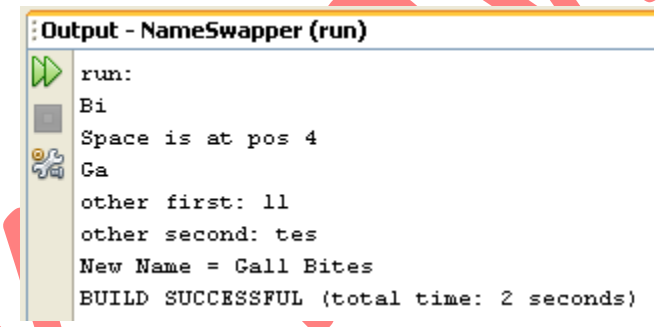
The Output is this:



We now have all the parts of the name. To join them together, we can use concatenation:

```
String NewName = "";
NewName = SurNameChars + OtherFirstChars + " " +
                         FirstNameChars + OtherSurNameChars;

System.out.println( "New Name = "  + NewName);
```

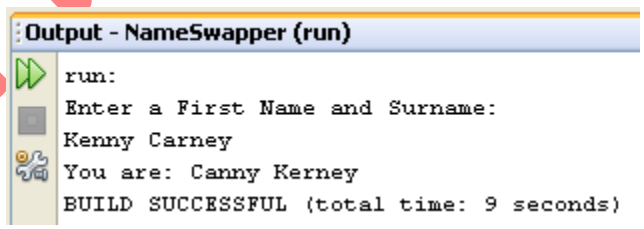
Add the new lines to your own code. When you run your programme, the Output window should display the following:



We can get rid of the print lines, though, and invite a user to enter a first name and second name. Here's the new programme (the only addition is for the keyboard input, which you've used before):

```
package nameswapper;
import java.util.Scanner;

public class NameSwap {

 public static void main(String[] args) {

  Scanner user_input = new Scanner(System.in);

  System.out.println( "Enter a First Name and Surname:" );
  String FullName = user_input.nextLine();

  String FirstNameChars = "";
  FirstNameChars = FullName.substring(0, 2);

  int spacePos = FullName.indexOf(" ");
  String SurNameChars = "";
  SurNameChars = FullName.substring(spacePos + 1,
                                    (spacePos + 1) + 2);
  String OtherFirstChars = "";
  OtherFirstChars = FullName.substring(2, spacePos);

  String OtherSurNameChars = "";
  OtherSurNameChars = FullName.substring((spacePos + 1) + 2,
                                         FullName.length());

  String NewName = "";
  NewName = SurNameChars + OtherFirstChars + " " +
                           FirstNameChars + OtherSurNameChars;
  System.out.println( "You are: "  + NewName);
 }
}
```

The Output window should look something like this, when you run your programme and enter a first name and surname:



We should, of course, add some error checking. But we'll assume the user can enter a first name and surname with a space between the two. If not, the programme will crash!

The equals Method in Java

You can check two strings to see if they are the same. For this, use the equals method in Java. Here's some code:

```
public static void main(String[] args) {

    String email_address1 = "meme@me.cob";
    String email_address2 = "meme@me.com";
    Boolean isMatch = false;

    isMatch = email_address1.equals(email_address2);

    if (isMatch == true) {
        System.out.println( "Email Address Match ");
    }
    else {
        System.out.println("Email addresses don't match");
    }
}
```

In this code, we want to check if one email address is the same as another. The first two lines set up two string variables, one for each email address. The third line sets up a Boolean variable. That's because the equals method returns a value of true or false. The fourth line is where we use the method:

isMatch = email_address1.equals(email_address2);

In between the round brackets of the **equals** method, you place the string you're trying to check. The other string goes before the equals method. Java will then tell you (true or false) whether the two are the same. The IF statement checks which one it is.

The equals method only compares objects, however. It's OK for strings, because they are objects. But you can't use the equals method to compare int variables. For example, this code would get you an error:

int num1 = 12;
int num2 = 13
Boolean isMatch = false;

isMatch = num1.equals(num2);

The int variable is a primitive data type, and not an object. You can turn the primitive int data type into an object, though:

```
int num1 = 12;
Integer num_1 = new Integer(num1);
```

Here, the int variable called num1 is being turned into an Integer object. Note the use of the new keyword. In between the round brackets of Integer, you put the primitive int data type you want to convert to an object.

The charAt method in Java

You can check to see which single character is in a particular string. The **charAt** method is used for this in Java. Here's some code to try:

```
String email_address = "meme@me.com";

char aChar = email_address.charAt( 4 );
System.out.println( aChar );
```

This code checks which letter as at position 4 in the email address string. The return value is a variable of type **char**:

```
char aChar = email_address.charAt( 4 );
```

When the above code is run, the output is the character @. The number between the round brackets of charAt is the position in the string you're trying to check. Here, we want to get at the character in position 4 of the email_address string. Again, the count starts at 0, just like substring.

One good use for charAt is for taking a letter from a string variable that is typed by a user, and then converting it to a single char variable. For example, you could ask the use to type Y to continue or an N to exit. Have a look at this code:

```
public static void main(String[] args) {

    Scanner user_input = new Scanner(System.in);

    System.out.println( "Quit Y/N");

    String aString  = user_input.next();

    char aChar = aString.charAt(0);

    if (aChar == 'Y') {
        System.out.println( "OK, BYE BYE");
    }
    else {
        System.out.println( "Not Quitting");
    }
}
```

We can't use the Scanner class directly to get a single letter to store in a char variable. So we use the next( ) method to get the next string that the user inputs. There's a next integer, next long, next double - even a next Boolean. But there's no next char. Even if the user inputs a single character it will still be a string and not a char. (Remember: a char variable stores a Unicode number as an integer.)

We can use charAt to get a character from any string that the user inputs, even if the user inputs a single letter:

char aChar = aString.charAt(0);

All we're saying is "Get the character at position 0 in the string called aString, then store it in the aChar variable".

We've added an IF statement to test what is in the aChar variable. (Note the use of single quotes around the letter Y.)

The replace Method in Java

The replace method in Java is used to replace all occurrence of a character/s in a particular string. Take this sentence:

"Where are you books?"

We want to replace "you" with "your". Here's the code:

```
public static void main(String[] args) {

    String aString = "Where are you books?";

    String amend = aString.replace("you", "your");

    System.out.println( amend );
}
```

There are several ways to use the replace method, and they differ in what you put between the round brackets of the method. We're replacing one sequence of characters with another. Think of the comma separating the two as the word "with". You'd then have "Replace you with your".

You can also replace single character:

aString.replace('£', '@')

The above code reads "Replace £ with @".

(You can also use something called a regular expression in your replace methods, but that is outside the scope of this book.)

Trim

You can trim white space from your strings White space is things like space characters, tabs, and newline characters - the characters you can't see, in other words. The trim method is easy to use:

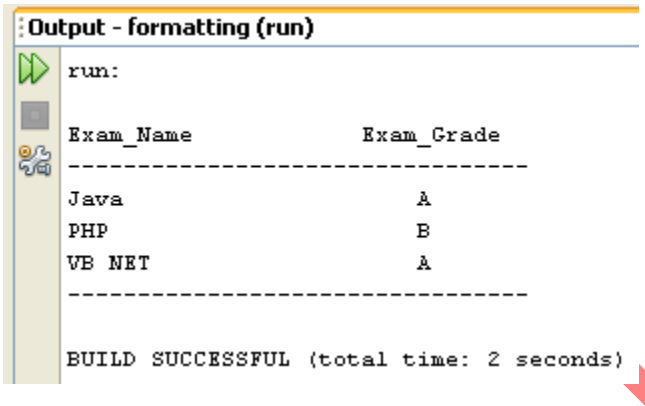
String amend = " white space ";
amend = amend.trim( );

So the trim method goes after the string you want to amend. The blank characters before the word "white" and after "space" in the code above will then be deleted.

If you're getting input from a user then it's always a good idea to use trim on the inputted strings.

Formatted Strings

Strings of text can be formatted and output with the **printf** command. The printf command understands a series of characters known as a **format specification**. It then takes a string of text and formats it, based on the format specification passed over. As an example, supposed we wanted the Output window to display text in neat columns, like this:



The first column is left-justified, and the second column is right-justified. The code for the Exam_Name and Exam_Grade headings was this:

String heading1 = "Exam_Name";
String heading2 = "Exam_Grade";

System.out.printf("%-15s %15s %n", heading1, heading2);

To get the left-justified column, you need a percentage symbol, a minus symbol, the number of characters, and then the letter "s" (lowercase). So "%-15s" means fifteen characters left-justified.

To get a right-justified column the same sequence of characters are used, except for the minus sign.

To get a newline %n is used. Note that the characters are surrounded with double quotes.

After a comma, you type the text that you want formatting. The first comma in the code above separates the format specification from the text being formatted.

Here's some tables of the various options.

String Formatting

"%s"	Format a string with as many characters as are needed
"%15s"	Format a string with the specified number of characters, and right-justify
"%-15s"	Format a string with the specified number of characters, and left-justify

If you want to format numbers then you can either use the "d" character or, for floating point numbers, the "f" character.

Integer Formatting

"%d"	Format a string with as many numbers as are needed
"%4d"	Format a string with the specified number of integers. Will pad with spaces to the left if not enough integers.
"%04d"	Format a string with the specified number of integers. Will pad with zeros to the left if not enough integers.

Floating Point Number Formatting

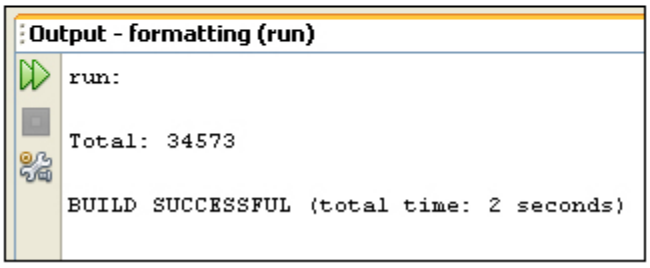
"%f"	Format a string with as many numbers as are needed. Will always give you six decimal places.
"%.2f"	Format a string with as many numbers as are needed. Gives 2 decimal places.
"%10.2f"	Format to 2 decimal places, but the whole string occupies 10 characters. If there's not enough numbers, then spaces are used to the left of the numbers.

Here are some code examples of String, integer and floating point formatting. Try them out for yourself.

```
System.out.printf( "%s %d %n", "Total:", 34573);
```

The text "Total:" will be formatted as a string (%s), while the numbers 34573 will be formatted as numbers (%d).

Output Window:



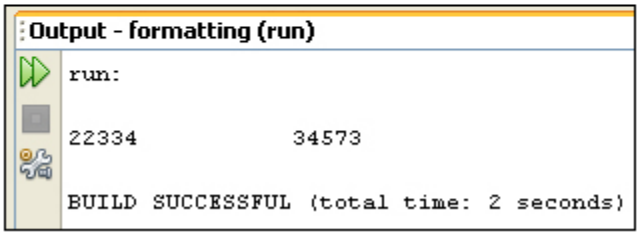
```
System.out.printf( "%s %10d %n", "Total:", 34573);
```

Same as above but the numbers occupy 10 places, with spaces to the left as padding.

```
System.out.printf( "%-10d %10d %n", 22334, 34573);
```

Two sets of numbers. The first one is left-justified; the second one is right-justified.

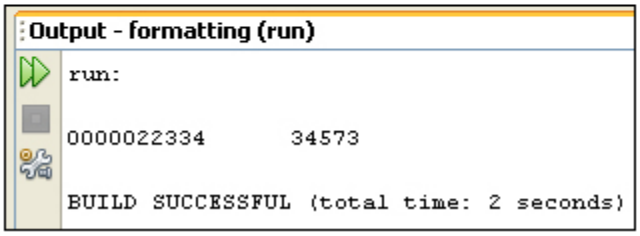
Output Window:



```
System.out.printf( "%010d %10d %n", 22334, 34573);
```

Two sets of numbers again. The first one is padded with leading zeros, this time. The second one is right-justified, but spaces are used as padding to the left instead of zeroes.

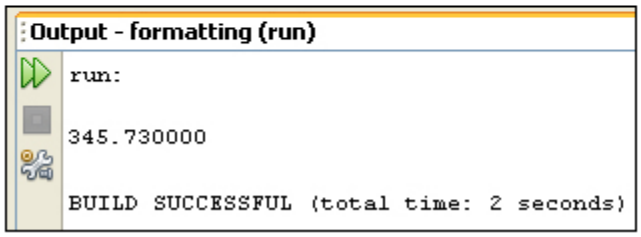
Output Window:



```
System.out.printf( "%f  %n",  345.73);
```

Format a floating point number, and add a newline character. The floating point number will have six decimal places.

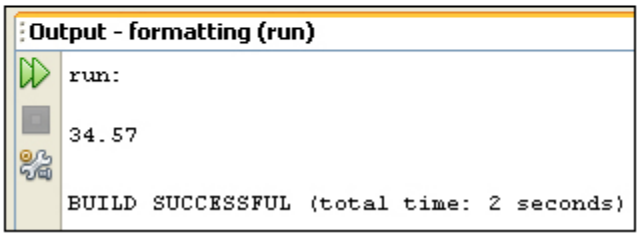
Output Window:



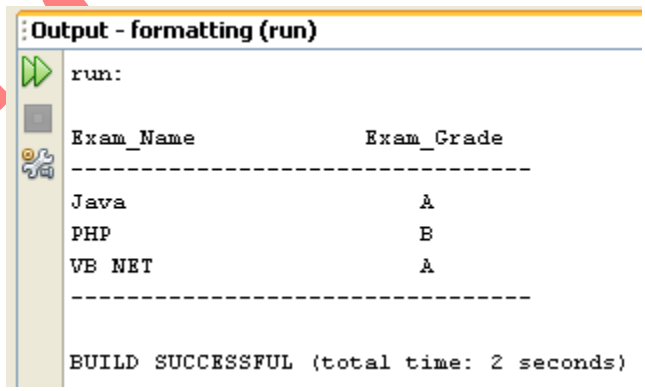
```
System.out.printf( "%.2f  %n",  34.573);
```

Same as above but will format to only two decimal places.

Output Window:



Finally, here's the table again from the start of this formatting section:



And here's the code for the above formatted output:

```
public static void main(String[] args) {

    String heading1 = "Exam_Name";
    String heading2 = "Exam_Grade";
    String divider = "--------------------------------";

    String course1 = "Java"; String grade1 = "A";
    String course2 = "PHP"; String grade2 = "B";
    String course3 = "VB NET"; String grade3 = "A";

    System.out.println("");
    System.out.printf("%-15s %15s %n", heading1, heading2);
    System.out.println(divider);

    System.out.printf("%-15s %10s %n", course1, grade1);
    System.out.printf("%-15s %10s %n", course2, grade2);
    System.out.printf("%-15s %10s %n", course3, grade3);

    System.out.println(divider);
    System.out.println("");

}
```

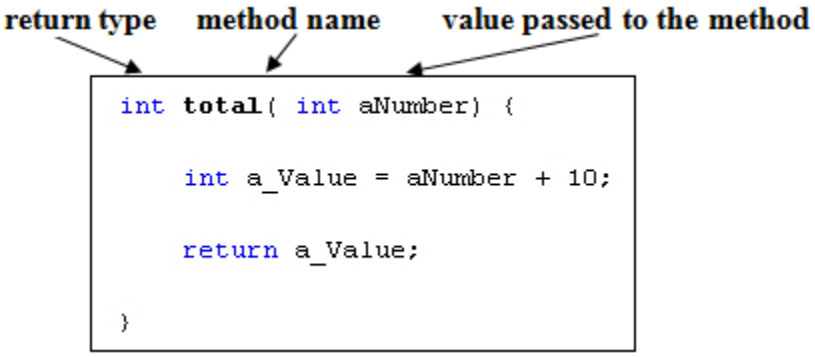
Have a play around with formatting, and see how you get on. If you get error messages you may have gotten your "s" formatting confused with your "d" formatting!

Writing you own Java Methods

You have been using methods in the previous section, and have seen how useful the inbuilt ones can be. In this section, you'll learn to write your own methods.

The Structure of a Method

A method is just a chunk of code that does a particular job. But methods are set out in a certain way. You have a method header, and a method body. The header is where you tell Java what value type, if any, the method will return (an int value, a double value, a string value, etc). As well as the return type, you need a name for your method, which also goes in the header. You can pass values over to your methods, and these go between a pair of round brackets. The method body is where you code goes.



The method's **return type** goes first, which is an **int** type in the code above. After the method type, you need a space followed by the name of your method. We've called the one above **total**. In between a pair of round brackets we've told Java that we will be handing the method a variable called **aNumber**, and that it will be an integer.

To separate this method from any other code, you need a pair of curly brackets. Your code for the method goes between the curly brackets. Note the word **return** in the method above. This is obviously the value that you want to return from your method, after your code has been executed. But it must be of the same type as the return type in the method header. So the return value can't be a string if you started the method with **int total**.

Sometimes you don't want Java to return anything at all. Think of Trim in the previous section. You may only want the Trim method to get on with its job, and not return anything to you. A method that doesn't return any value at all can be set up with the word **void**. In which case, it doesn't need the **return** keyword. Here's a method that doesn't return a value:

```
void print_text(String someText) {

    System.out.println( "Some Text Here" );

}
```

All the method above does is to print some text. It can just get on with its job, so we've set it as a void method. There's no return value.

Methods don't need to have values passed to them. You can just execute some code. Here's a void method without any values being passed over:

```
void print_text() {

    System.out.println( "Some Text Here" );

}
```

And here's an **int** method that has no values being passed:

```
int total() {

    int a_Value = 10 + 10;

    return a_Value;

}
```

As you can see, the round brackets are empty in both methods. But they are still needed. Miss the round brackets out and you'll get an error message.

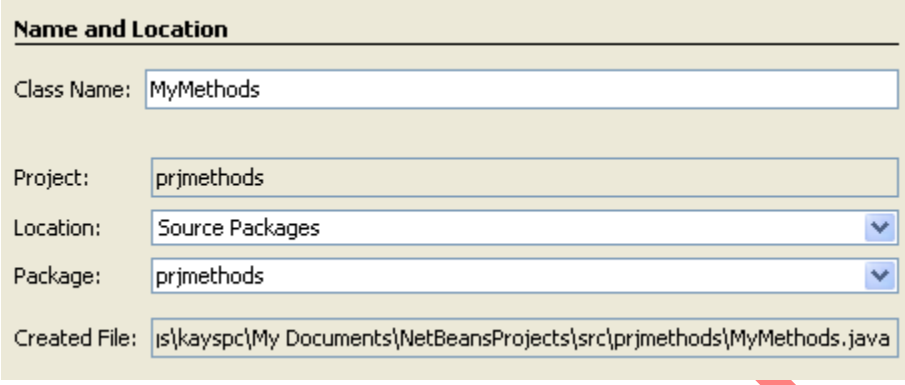
Java Method Calling

Methods don't do anything until you call them into action. Before we see how, let's add another class to the project. We can then put all the methods there, instead of clogging up the main class. (You'll start learning more about classes in the next section.)

Start a new Java Application project. Give your project a name, and rename the Main method to something else. Then click Finish. In the image below, we've called our project **prjmethods**, and the class **TestMethods**:

```
package prjmethods;

public class TestMethods {

    public static void main(String[] args) {


    }

}
```

To add a new class to your project, click **File** from the NetBeans menu. From the File menu, select **New File**. You'll see a dialogue box appear. In the **Categories** section select **Java**, and in the **File Types** section select **Java Class**. Then click the Next button at the bottom. In step two, type a name for your new class. We've called ours **MyMethods**. You can leave everything else on the defaults:



So we're creating a second class called **MyMethods** which will be in the Project **prjmethods**. Click the Finish button and your new class file will be created. A new tab will appear in the NetBeans software, with some default comments about how to change templates. You can delete these comments, if you like. You should be left with the following code window:

```
package prjmethods;


public class MyMethods {


}
```

The thing to notice is that there's no Main method this time - just a blank class with the name you chose, and a pair of curly brackets for your code. Let's add one of our methods. So add the following code to your class:

```
package prjmethods;

public class MyMethods {

    int total() {
        int a_Value = 10 + 10;

        return a_Value;
    }

}
```

This is the **int** method we met earlier with the name total. It has nothing between the round brackets, which means we're not going to be handing it any values. All the method does is to add up 10 + 10 and store the answer in a variable called **a_Value**. This is the value that will be returned from the method. The value after the keyword return must match the return type from the method header. Ours is OK because they are both **int**.

(It's important to bear in mind that the **a_Value** variable can't be seen outside of the total method: Any variable set up inside of a method can't be accessed outside of that method. It's known as a **local** variable - it's local to the method.)

To call the total method, select your TestMethods tab in NetBeans, the one with your Main method. We're going to call the total method from the Main method.

The first thing to do is to create a new object from our MyMethods class. Add the following line to your Main method:

```
package prjmethods;

public class TestMethods {

    public static void main(String[] args) {

        MyMethods test1 = new MyMethods();

    }

}
```

To create a new object from a class you start with the name of the class, **MyMethods** in our case. This is in place of int, double, String, etc. In other words, the type of variable you're creating is a MyMethods variable. After a space, you type a name for your new MyMethods variable. We've called ours **test1**. (It's underlined because we haven't done anything with it yet. This is a NetBeans underline.)

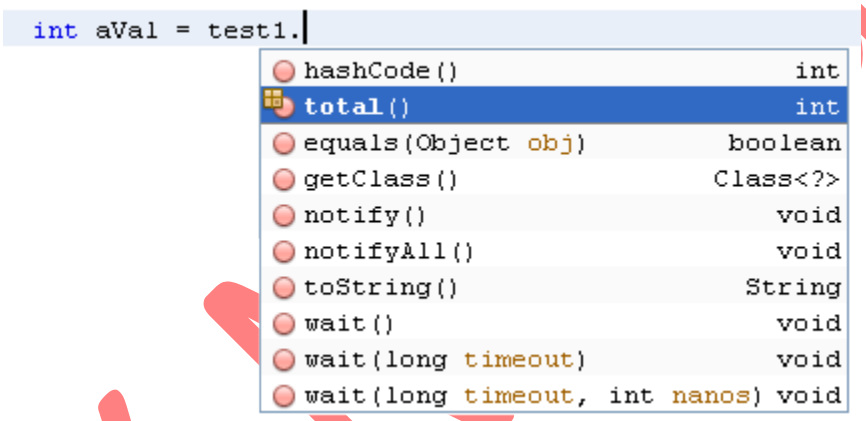
An equals sign comes next, followed by the keyword **new**. This means **new object**. After the keyword **new**, type a space followed by your class name again. The class name needs a pair of round brackets after it, this time. End the line in the usual manner, with a semi-colon.

What we've done here is to create a new MyMethods object with the name test1. Our **total** method inside of the class **MyMethods** will now be available from the Main method of the TestMethods class.

To call the total method into action, add this line:

```
public class TestMethods {

    public static void main(String[] args) {

        MyMethods test1 = new MyMethods();

        int aVal = test1.total();

    }
}
```

We're setting up an **int** variable with the name **aVal**. After an equals sign comes the name of our class, test1. To access methods in the class, type a dot. NetBeans will display a popup box with the available methods:



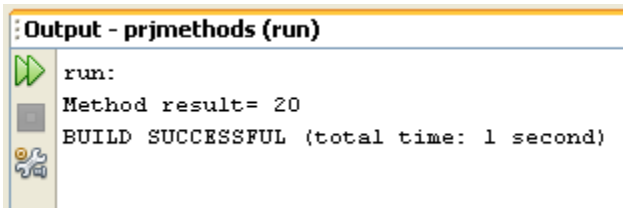
Our total variable is on the list (the others are built in methods). The round brackets are empty because our method doesn't accept values, but the return type, int, is displayed to the right.

Double click total to add it to your code. Then type a semi-colon to end the line.

Finally, add a print line:

```
public class TestMethods {

    public static void main(String[] args) {

        MyMethods test1 = new MyMethods();

        int aVal = test1.total();

        System.out.println( "Method result= " + aVal );

    }
}
```

When the code is run, the Output window will display the following:



So to call a method that returns a value, note what value is being returned by your method. Then assign this value to a new variable, **aVal** in our case. But the method should be available when you type a dot after your object name.

If your method is of type void, however, you don't need to assign it to a new variable like a Val. As an example, switch back to your MyMethods class, and add the void method you met earlier:

```
package prjmethods;

public class MyMethods {

     int total() {
        int a_Value = 10 + 10;

        return a_Value;
    }

     void print_text() {

        System.out.println( "Some Text Here" );
    }

}
```

This new method is called **print_text**. Again, it has an empty pair of round brackets as we're not handing it any values. All it does is to print out some text.

Once you've added the void method, switch back to your TestMethods class. Add the following line:

test1.print_text( )

As soon as you type the dot, you should see the new method appear on the list:

```
System.out.println( "Method result= " + aVal );

test1.
```

```
equals(Object obj)          boolean
getClass()                  Class<?>
hashCode()                  int
notify()                    void
notifyAll()                 void
print_text()                void
toString()                  String
total()                     int
wait()                      void
wait(long timeout)          void
wait(long timeout, int nanos) void
```

Both of our methods are now on the list, **total** and **print_text**. The values that they return are displayed on the right, **int** and **void**.

Because the print_text method is a void method, you don't need to set up a return value. All you need is the name of your object, a dot, and the void method you want to call. Java will then just get on with executing the code inside of your method.

Run your code and your Output window should display the following:

```
Output - prjmethods (run)
run:
Method result= 20
Some Text Here
BUILD SUCCESSFUL (total time: 2 seconds)
```

Passing Values to your Java Methods

You can pass values to your methods so that something can be done with this value. That value goes between the round brackets of the method.

Switch back to your MyMethods class. Now add the second of our total methods:

```
package prjmethods;

public class MyMethods {

     int total() {
        int a_Value = 10 + 10;

        return a_Value;
    }

     void print_text() {

        System.out.println( "Some Text Here" );
    }

     int total(int aNumber) {
        int a_Value = aNumber + 20;

        return a_Value;
    }
}
```

We now have two methods with the same name: **total**. The difference between the two is that this new one has a value between the round brackets. Java allows you to do this, and it is called **method overloading**. You can have as many methods with the same name as you want, with any return value. However, you can't have the same type of variables between the round brackets. So you can't have two total methods that return int values with both of them having int values between the round brackets. You can't do this, for example:

```
int total( int aNumber ) {
int a_Value = aNumber + 20;

return a_Value;
}

int total( int aNumber ) {
int a_Value = aNumber + 50;

return a_Value;
}
```

Although the two methods do different things, they have the same method headers.

Before you try out your new method, add some comments directly above the method:

```
/**
 * Returns an integer value, which is
 * 20 plus the number passed as a paramater.
 *      @param aNumber  Any Integer value
 *      @return 20 + the value of aNumber
 */
int total(int aNumber) {
    int a_Value = aNumber + 20;

    return a_Value;
}
```

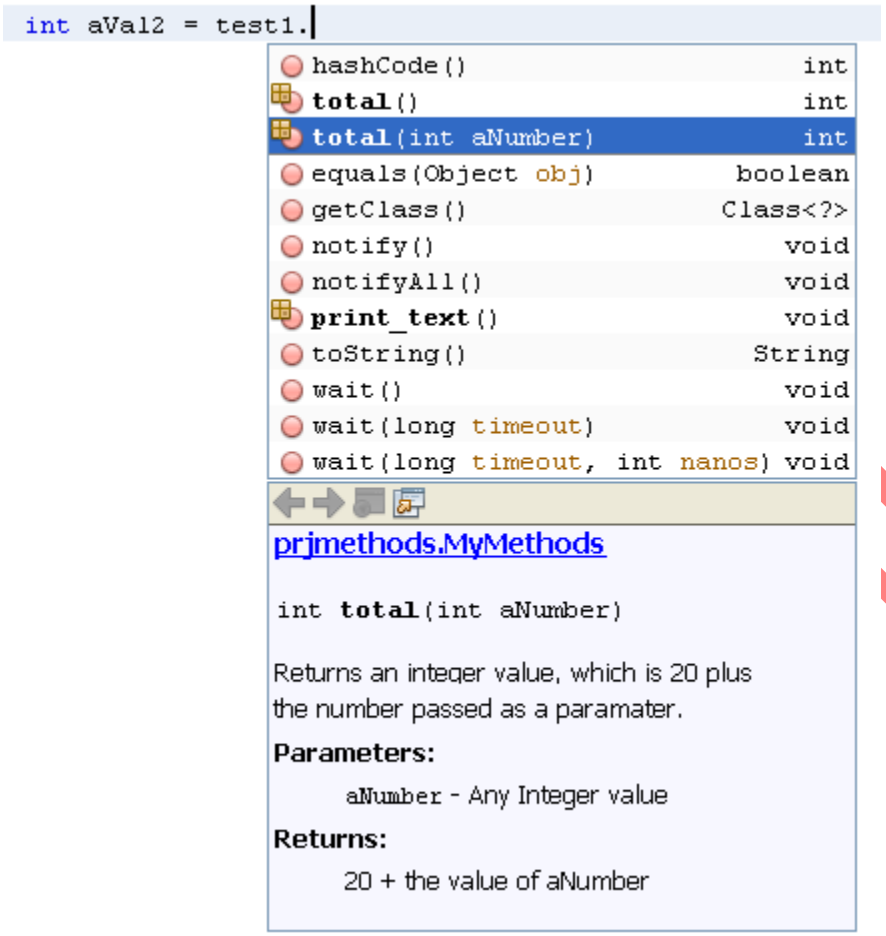
You'll see what the comments do in a moment. But the **param** in the comments above is short for **parameter**. A parameter is the technical term for the value between the round brackets of your method headers. Our parameter is called **aNumber**, and it has an integer values. Note the use of the @ character before **param** and **return**.

All we're doing with the method itself is passing it an integer value and adding 20 to this passed value. The return value is the total of the two.

Now switch back to your code and add the following line:

int aVal2 = test1.total(30);

As soon as you type the dot after your **test1** object, you'll see the popup list again. Your new total method will be on it. Click on the new method to highlight it and NetBeans will display the following:

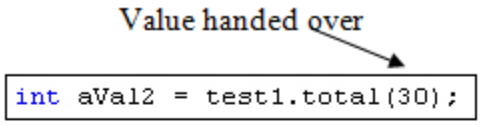


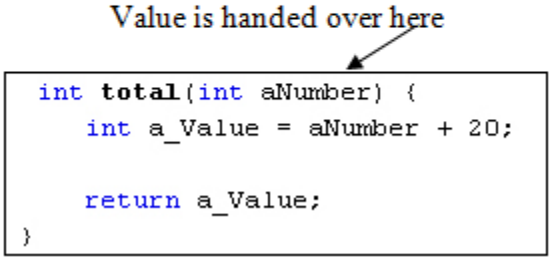
The comments that we added are now displayed in the blue box underneath the list of methods. Anyone else coming across your method should be able to figure out what it does. The @param and @return lines from the comments are now filled out, and made bold.

But once you have added the total2 method, type the number 30 between the round brackets. Then type a semi-colon to end the line. You main method should now look like this:

```
public static void main(String[] args) {

    MyMethods test1 = new MyMethods();

    int aVal = test1.total();

    System.out.println( "Method result= " + aVal );

    test1.print_text();

    int aVal2 = test1.total(30);

}
```

The 30 between the round brackets of the new total method will be handed over and placed in the variable aNumber, which is in the round brackets of the method header:



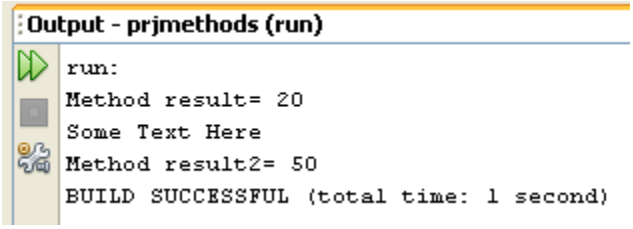


Once you have handed the value over, the method can get to work.

Add a print line to your code:

System.out.println("Method result2= " + aVal2);

Then run your programme. The Output window should display the following:



So our new total method has added 30 to 20, and then returned the answer to the variable called aVal2.

Passing Multiple Values to Methods

You can pass more than one value over to your methods. Add the following method to your MyMethods class:

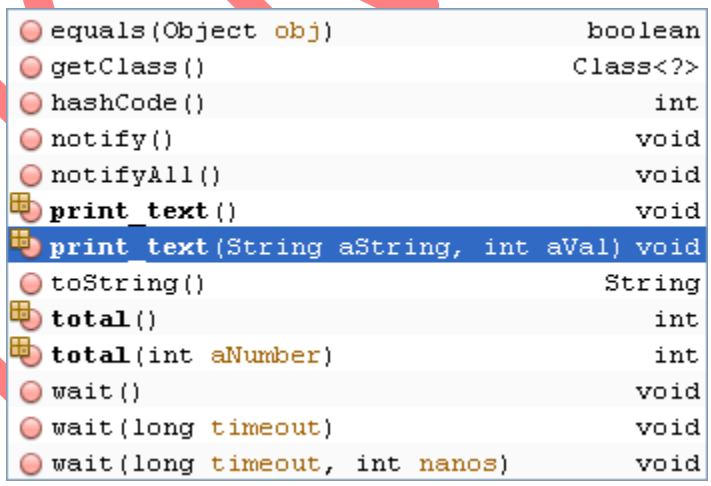
```
void print_text(String aString, int aVal) {

    System.out.println( aString  + aVal );

}
```

All this method does is to print something out. Between the round brackets of the method name we have two values, a String variable called aString and an int variable called aVal. When we call this method, we need a string first and then a number. Try to do it the other way round and you'll get error messages.

Go back to your TestMethods class and make the following call to the method:

test1.print_text("The value was ", aVal2);

Again, the **print_text** method should show up on the NetBeans popup list.



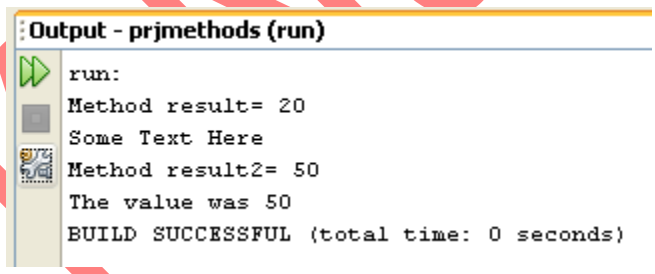
The values (parameters) that we specified are between the round brackets, along with the return method, void.

But your main coding window should now look like this:

```
public static void main(String[] args) {

    MyMethods test1 = new MyMethods();

    int aVal = test1.total();

    System.out.println( "Method result= " + aVal );

    test1.print_text();

    int aVal2 = test1.total(30);
    System.out.println( "Method result2= " + aVal2 );

    test1.print_text("The value was ", aVal2);

}
```

The two values that we are passing over are separated by a comma. Notice that the value inside of **aVal2** is being passed over. However, the variable name inside of the round brackets of **print_text** is called **aVal**. Although the two variable names are different, this doesn't matter. What you are doing is passing values over to the method. So the variable aVal will end up with the same value as aVal2.

If you run your programme now, you should see the following in the Output window:



Java Classes

In this section, you'll learn how to write your own Java classes, and how to create objects from them. You've already made a start on classes in the previous section. But we'll go into more detail now. In case you're confused about the difference between an object and class, though, when we talk about a class we're talking about the code itself, the code that sits around doing nothing. When you put the code to work it's an object.

When you create a class, you're writing code to do a particular job. That job might be to do with an employee, but not the company's sales figures at the same time. You would write a separate class for the sales figures. That way, you can re-use the employee class in another project. The sales figures would be redundant data.

When you're trying to come up with your own ideas for classes you should bear in mind that redundancy issue and ask yourself, "Is there any code in this class that doesn't need to be here?"

Create a new Java project for this. Call the package **exams**, and then change the name of the Main method to **ExamDetails**. You should then have the following code:

```
package exams;

public class ExamDetails {

    public static void main(String[] args) {

    }

}
```

We'll create a second class to handle the exam data. So, in NetBeans, click **File** from the menu bar at the top. From the File menu, select **New File**. Highlight **Java** from the Categories list, and **Java Class** from the File Types list. Then click Next. On the next screen, enter **StudentResults** as the class name. Then click Finish. NetBeans will create a second class in your project. You can delete any default comments.

Java Field Variables

n a previous section, we talked about variables inside of methods. The variables you set up inside of methods are only available to those methods. They are not available to other methods. They are said to have local scope.

However, you can set up variables outside of methods that all the methods in your class can see. These are known as Field variables (or Instance variables). You set them up in exactly the same way as any other variable. Add the following four fields to your new StudentResults class:

```
package exams;

public class StudentResults {

    String Full_Name;
    String Exam_Name;
    String Exam_Score;
    String Exam_Grade;

}
```

We're setting up four string variables (four string fields). As the names of the fields suggest, the string will hold a person's name, an exam name, a score, and a grade. These four fields will be

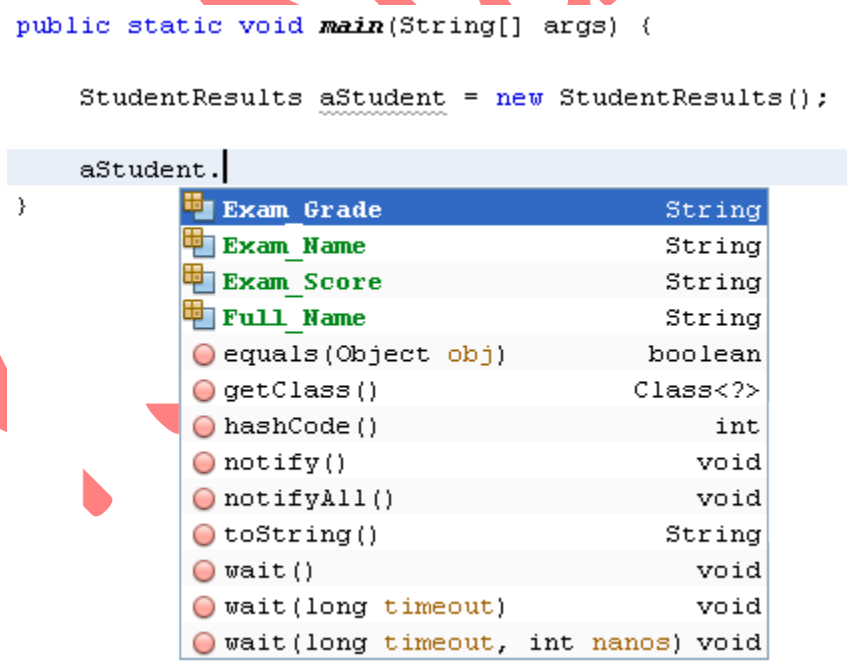
available to all the methods that we write in this class, and won't be local to any one method. They are said to have global scope.

To see just how global they are, click back on to your ExamDetails class, the one with the main method. Add the following line of code to create a new object from your StudentResults class:

```
package exams;

public class ExamDetails {

    public static void main(String[] args) {

        StudentResults aStudent = new StudentResults();

    }

}
```

This is the same as we did in the previous section - used the new keyword to create a new object. The name of the object will be **aStudent**, and it is of type StudentResults, which is our class.

On the next line, type the name of the variable (aStudent), followed by a dot. As soon as you type the dot NetBeans display a popup list of methods and properties available to your object:



The four fields that we set up are on the list. They are not methods, but something called a property. The fact that they are on the list at all means that they have global scope. If they had local scope they wouldn't be on the list.

You can set values for properties. Try this: add the following highlighted code to your main method:

```
public class ExamDetails {

    public static void main(String[] args) {

        StudentResults aStudent = new StudentResults();

        aStudent.Exam_Name = "VB NET";

        String exam = aStudent.Exam_Name;
        System.out.println( exam );
    }

}
```

We've selected the **Exam_Name** field from the list and assigned it the value "VB NET". The next line then gets the value of Exam_Name from the aStudent object. The result is stored in a variable called exam, and then printed out. When you run the programme, the output is the string "VB Net".

So the four variables we've set up are now available to both classes.

However, it's not a good idea to make field variables global like this. You tend to lose track of just what values are being held in them, and it therefore makes debugging your code a lot harder. It's considered good coding to narrow the scope of field variables.

To make a field variable available only to a particular class, you add the Java keyword **private** just before the field declaration. Change the code in your StudentResults class to this:

```
package exams;

public class StudentResults {

    private String Full_Name;
    private String Exam_Name;
    private String Exam_Score;
    private String Exam_Grade;

}
```

Now, only code within the StudentResults class can see these field variables. To check, go back to your main method. You should see some warnings and red underlines:

```
    public static void main(String[] args) {

        StudentResults aStudent = new StudentResults();
                          Exam_Name has private access in exams.StudentResults
        aStudent.Exam_Name = "VB NET";

        String exam = aStudent.Exam_Name;
        System.out.println( exam );
    }
```

Delete the three lines at the bottom. Type **aStudent** and then a dot again to see the NetBeans popup list:

```
aStudent.
```

equals(Object obj)	boolean
getClass()	Class<?>
hashCode()	int
notify()	void
notifyAll()	void
toString()	String
wait()	void
wait(long timeout)	void
wait(long timeout, int nanos)	void

As you can see, the four field variables have now vanished. They have vanished because they no longer have global scope, and therefore can't be seen from the ExamDetails class.

The Java Class Constructor

Because we've made the field variables private, we need another way to assign values to them. One way to do this is with something called a **constructor**. This is a method you can use to set initial values for field variables. When the object is created, Java calls the constructor first. Any code you have in your constructor will then get executed. You don't need to make any special calls to a constructor method - they happen automatically when you create a new object.

Constructor methods take the same name as the class. Add the following constructor to your StudentResults class:

```
package exams;

public class StudentResults {

    private String Full_Name;
    private String Exam_Name;
    private String Exam_Score;
    private String Exam_Grade;

    StudentResults() {

    }

}
```

So the name of the constructor is **StudentResults**. This is exactly the same name as the class itself. Unlike normal methods, class constructors don't need a return type like int or double, nor any return value. You can, however, pass values over to your constructors. If we want to pass values over to our field variables, we can do this:

```
public class StudentResults {

    private String Full_Name;
    private String Exam_Name;
    private String Exam_Score;
    private String Exam_Grade;

    StudentResults( String name, String grade ) {

        Full_Name = name;
        Exam_Grade = grade;
    }
```

Here, we've added two String variables to the round brackets of the constructor. Inside the curly brackets we've assigned these values to the **Full_Name** and **Exam_Grade** fields. When you create a new object, you'd then need two strings between the round brackets of the class name:

StudentResults aStudent = new StudentResults("Bill Gates", "A");

When the object is created, the values "Bill Gates" and "A" will be handed over to the constructor.

However, it's a good idea to just set some default values for your field variables. These values will then be assigned when the object is created. Add the following to your constructor:

```
package exams;

public class StudentResults {

    private String Full_Name;
    private String Exam_Name;
    private String Exam_Score;
    private String Exam_Grade;

    StudentResults() {
        Full_Name = "No Name Given";
        Exam_Name = "Unknown";
        Exam_Score = "No Score";
        Exam_Grade = "Unknown";
    }

}
```

All four of our field variables will now have default values whenever a new StudentResults object is created. Notice that we now don't have anything between the round brackets of the class constructor.

Accessing Class Variables

Now that we have some default values, we can add a method that sets some different values for them. Add the following method to your StudentResults class:

```
public class StudentResults {

    private String Full_Name;
    private String Exam_Name;
    private String Exam_Score;
    private String Exam_Grade;

    StudentResults() {
        Full_Name = "No Name Given";
        Exam_Name = "Unknown";
        Exam_Score = "No Score";
        Exam_Grade = "Unknown";
    }

    String fullName(String aName) {

        Full_Name = aName;
        return Full_Name;
    }

}
```

This new method is called **fullName**, and has a String variable called **aName** between its round brackets. The method doesn't do a great deal, and is here for simplicity's sake. We could have had the method do more work, like checking it for errors, making sure it's proper case, checking for blank strings, etc. But the important point is that it sets a value for the **Full_Name** field, and returns this field as a value. When we call this method, it will overwrite the default value for Full_Name and insert a new value. Whatever is in the variable **aName** will be the new Full_Name value. Let's see it in action.

Click back to your ExamDetails class. Add the following two lines:

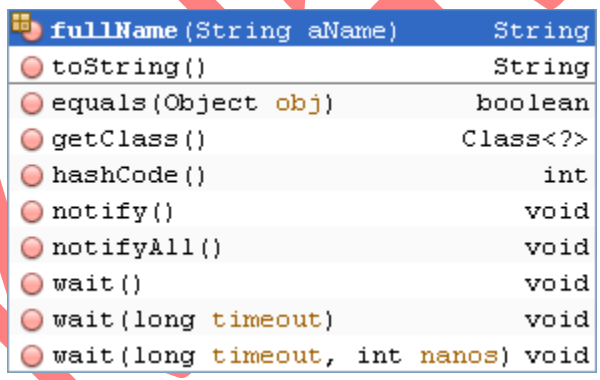
String sName = aStudent.fullName("Bill Gates");
System.out.println(sName);

The code for your ExamDetails class should look like the one below:

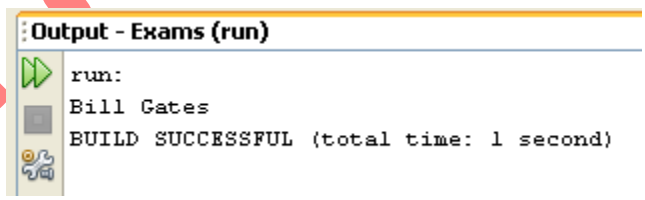
```
package exams;

public class ExamDetails {

    public static void main(String[] args) {

        StudentResults aStudent = new StudentResults();

        String sName = aStudent.fullName("Bill Gates");
        System.out.println( sName );

    }
}
```

What we're doing here is calling the **fullName** method of our **aStudent** object. We're handing over a value of "Bill Gates". This will then be the value for the **Full_Name** field. (It could have been checked for errors, amended, and then stored in the field.) The Full_Name value is then returned and stored in the variable **sName**.

Just like the methods we created in the previous section, though, our fullName method is on the NetBeans popup list. Notice that the constructor is not there, however:



Run your code to test it out. The Output window should display the following:



What we've done, then, is to set a value for a field variable in a class called **StudentResults**. We've then accessed that value and printed it out.

Let's now add a method that actually does something useful. What we'll do is to allow a user to enter a two letter exam code. We'll then pass those two letters over to a method that turns the two

letters into an exam name. For example, if a user enters the letter "VB", the method will return the string "Visual Basic .NET". The longer string will be stored in the Exam_Name field variable.

Add the following code to your StudentResults class, just below your fullName method:

```
String fullName(String aName) {

    Full_Name = aName;
    return Full_Name;
}

    String examName(String examCode) {

        if (examCode.equals("VB")) {
            Exam_Name = "Visual Basic .NET";
        }
        else if (examCode.equals("JV")) {
            Exam_Name = "Java";
        }
        else if (examCode.equals("C#")) {
            Exam_Name = "C# .NET";
        }
        else if (examCode.equals("PH")) {
            Exam_Name = "PHP";
        }
        else {
            Exam_Name = "No Exam Selected";
        }

        return Exam_Name;
}
```

The **examName** method has a string variable called examCode between its round brackets. This string will be the two letters. The IF … ELSE IF lines check to see which two letters are in the string. If we find a match for the two letters then a longer exam title is placed into the Exam_Name field. If no match is found then the text will be "No Exam Selected".

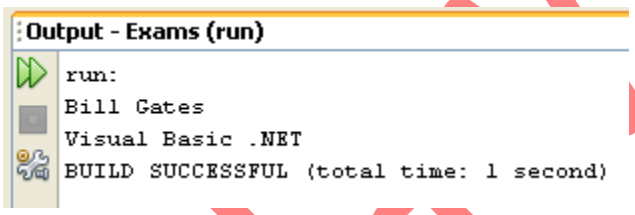
Go back to your ExamDetails class and add the following line:

String exam = aStudent.examName("VB");

Again, we're just making a call to the method. We hand over to the two letters "VB". The method returns the value "Visual Basic .NET", and then stores it in the string variable we've called exam. Add a new print line and your code should look like this:

```
public class ExamDetails {

    public static void main(String[] args) {

        StudentResults aStudent = new StudentResults();

        String sName = aStudent.fullName("Bill Gates");
        String exam = aStudent.examName("VB");

        System.out.println( sName );
        System.out.println( exam );

    }
}
```

Run your code and the Output should be this:



More Java Class Methods

So we now have a student name and an exam name. Both are stored in the field names in the StudentResults class. We can now add an exam score.

Add the following new method to your class, just after your examName method:

```
String examScore(int aScore){

    Exam_Score = aScore + " out of 50";
    return Exam_Score;
}
```

This new method has the name **examScore**, with an int variable called **aScore** between its round brackets. It's set to return a String value. The method itself just combines the score with the string " out of 50". So if the value in aScore is 30, the text "30 out of 50" will be stored in the Exam_Score field.

In your ExamDetails class, add the following line:

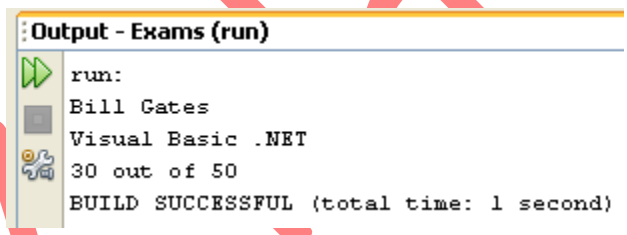
String score = aStudent.examScore(30);

So we call the new **examScore** method and hand it a value of 30. The value in the **Exam_Score** field is returned, and then stored in a string variable that we've called score.

Add a new print line method so that your code looks like ours below:

```
public class ExamDetails {

    public static void main(String[] args) {

        StudentResults aStudent = new StudentResults();

        String sName = aStudent.fullName("Bill Gates");
        String exam = aStudent.examName("VB");
        String score = aStudent.examScore(30);

        System.out.println( sName );
        System.out.println( exam );
        System.out.println( score );
    }
}
```

When the programme is run, the Output window now looks like this:



So we have the student name, the exam name, and the score out of 50. We can now add a grade to the output.

We'll use a single letter for the grade: A, B, C, D, or E. If the students gets 41 or over, we'll an award an A; if the score is between 31 and 40, the grade will be B; for a score of 21 to 30 a C grade will be awarded; a D grade is between 11 and 20; and an E is for scores between 0 and 10.

Add the following method to calculate the above grades (add it to your StudentResults class):

```
private String getGrade(int aScore) {

    String examGrade = "";

    if (aScore >= 0 && aScore <= 10) {
        examGrade = "E";
    }
    else if (aScore >= 11 && aScore <= 20) {
        examGrade = "D";
    }
    else if (aScore >= 21 && aScore <= 30) {
        examGrade = "C";
    }
    else if (aScore >= 31 && aScore <= 40) {
        examGrade = "B";
    }
    else if (aScore >= 41 ) {
        examGrade = "A";
    }
    return "Grade is " + examGrade;
}
```

Notice that this method is **private**. Just like field variables, making a method private means it can only be seen inside of this class. It can't be seen by the ExamDetails class.

To get the grade, we'll set up another method inside of the StudentResults class. We'll use this to get the grade. Add the following method just above the **getGrade** method (though you can add it below, if you prefer: it doesn't make any difference to Java!):

String examGrade(int aScore) {

Exam_Grade = this.getGrade(aScore) ;
return Exam_Grade;
}

This is the method we'll call from the ExamDetails class, rather than the getGrade method. The name of this new method is **examGrade**, and again we're passing in the student's score. Look at this line, though:

Exam_Grade = this.getGrade(aScore);

The getGrade method is being called, here, and we're passing it the score that was handed over. Calling one method from another is standard practice, and it allows you to simplify your code. The alternative is to have very long methods that are hard to read.

Another thing to notice in the line above is the Java keyword **this**. The **this** keyword means "this class", rather than another class that may have the same method name. It avoids any confusion. It's strictly not necessary, and we could have left it out. The method call would still work without it:

Exam_Grade = getGrade(aScore);

The end result, though, is still the same: we're storing something in the Exam_Grade field variable, and that something will be the text "Grade is" plus a grade letter.

Add the following line to your ExamDetails class, to test out your new methods:

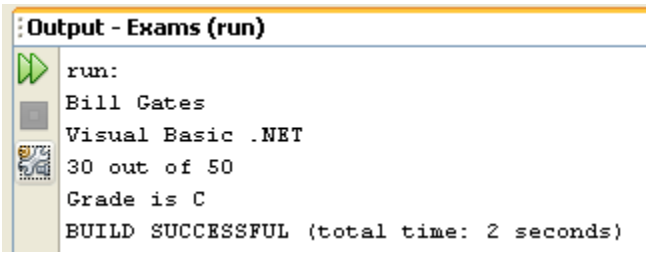
String grade = aStudent.examGrade(30);

This line hands a value of 30 over to the examGrade method. The value in the Exam_Grade field variable is then returned, and stored in a variable called grade.

With a print line, your ExamDetails class should look like this:

```
public class ExamDetails {

    public static void main(String[] args) {

        StudentResults aStudent = new StudentResults();

        String sName = aStudent.fullName("Bill Gates");
        String exam = aStudent.examName("VB");
        String score = aStudent.examScore(30);
        String grade = aStudent.examGrade(30);

        System.out.println( sName );
        System.out.println( exam );
        System.out.println( score );
        System.out.println( grade );
    }
}
```

Run your programme to see the Output window:



If your programme is not working properly, here's the full code for the **StudentResults** class:

```
package exams;

public class StudentResults {

    private String Full_Name;
    private String Exam_Name;
    private String Exam_Score;
    private String Exam_Grade;

    StudentResults() {
        Full_Name = "No Name Given";
        Exam_Name = "Unknown";
        Exam_Score = "No Score";
        Exam_Grade = "Unknown";
    }

    String fullName(String aName) {

        Full_Name = aName;
        return Full_Name;
    }

    String examName(String examCode) {

        if (examCode.equals("VB")) {
            Exam_Name = "Visual Basic .NET";
        }
        else if (examCode.equals("JV")) {
            Exam_Name = "Java";
        }
        else if (examCode.equals("C#")) {
            Exam_Name = "C# .NET";
        }
        else if (examCode.equals("PH")) {
            Exam_Name = "PHP";
        }
        else {
            Exam_Name = "No Exam Selected";
        }

        return Exam_Name;
    }
```

```
    String examScore(int aScore){

        Exam_Score = aScore + " out of 50";
        return Exam_Score;
    }

    String examGrade(int aScore) {
        Exam_Grade = this.getGrade(aScore);
        return Exam_Grade;
    }

    private String getGrade(int aScore) {

        String examGrade = "";

        if (aScore >= 0 && aScore <= 10) {
            examGrade = "E";
        }
        else if (aScore >= 11 && aScore <= 20) {
            examGrade = "D";
        }
        else if (aScore >= 21 && aScore <= 30) {
            examGrade = "C";
        }
        else if (aScore >= 31 && aScore <= 40) {
            examGrade = "B";
        }
        else if (aScore >= 41 ) {
            examGrade = "A";
        }
        return "Grade is " + examGrade;
    }

}
```

Java and Inheritance

Another important concept in Object Oriented programming is **Inheritance**. Just what Inheritance is will become clearer with some programming examples. But essentially, it's having one class as a parent class (called a **super** class) and another class as a child of the parent (the **sub** class). The child class is said to be derived from the parent class. The reason to have a child class, though, is to keep information separate. The child can inherit all the methods and fields from its parent, but can then do its own thing.

As an example of inheritance, we'll create a sub class that handles information about certificates. If the student gets an "A" grade, we'll award a Certificate of Excellence; if the student gets a "B"

or a "C" we'll award a Certificate of Achievement. Any other grade and no certificate will be awarded. But the point about the sub class is to keep the certificates data separate from the exam data. However, we may still want to access some of the information about the exam, such as which exam it was. We can even access the methods that turn a score into a grade, and all from the sub class.

So, create a new class by clicking **File > New File** from the NetBeans menu. When the dialogue box appears, select **Java** under the Categories heading, and **Java Class** under **File Types**. Click Next, and enter **Certificates** as the name of your new class. Click Finish.

When your new class is created, add a private String field and call it certificate. Your new class should then look like this:

```
package exams;

public class Certificates {

    private String certificate;

}
```

To create a **sub class** (child) from a Java **super class** (parent), the keyword **extends** is used. You then follow the "extends" keyword with the parent class you want to extend. We want to create a sub class from the StudentResults class. The StudentResults class will be the super class and the Certificates class will be the sub class.

After "public class Certificates" in your code add "extends StudentResults". Your code should then look like this:

```
package exams;

public class Certificates extends StudentResults {

    private String certificate;

}
```

You have now created a sub class that inherits the code from the StudentResults class.

Just like the StudentResults class we can create a constructor for this new Certificates class. When we create an object from the class, Java will first of all call our constructor.

However, only one constructor can be called. If we call a new constructor from the Certificates class all those default values we set up for the StudentResults class fields won't get set. To get

around this, there is a keyword called **super**. This makes a call to the constructor from the super class. Add the following constructor to your Certificates class:

```
package exams;

public class Certificates extends StudentResults {

    private String certificate;

    Certificates() {
        super();
        certificate = "No Certificate Awarded";
    }
}
```

The name of the constructor is the same as the name of the class: **Certificates**. The first line of the code between curly brackets is the super call (note the round brackets after super). When this line executes, all the default fields we set up in StudentResults will be set.

The second line of code in the constructor sets a default value for the String field called certificate. (You can actually set up more than one constructor. You'll see how to do this in a later section.)

To test out your new class, go back to your ExamDetails class, the one with the main method. Comment out any code you have so far. A quick way to do this is to highlight all the code, and then click the comments icon on the NetBeans toolbar:

To get rid of comments, highlight the code again and click the uncomment icon.

Now add the following line to create a new object from your class:

Certificates c1 = new Certificates();

Your code window should look something like this:

```
package exams;

public class ExamDetails {

    public static void main(String[] args) {

//        StudentResults aStudent = new StudentResults();
//        String sName = aStudent.fullName("Bill Gates");
//        String exam = aStudent.examName("VB");
//        String score = aStudent.examScore(30);
//        String grade = aStudent.examGrade(30);
//        System.out.println( sName );
//        System.out.println( exam );
//        System.out.println( score );
//        System.out.println( grade );

          Certificates c1 = new Certificates();

    }
}
```

The object name is simply **c1**. The type of object is a Certificates object.

To check that you can access the methods from the StudentResults class, add this line below you new c1 object:

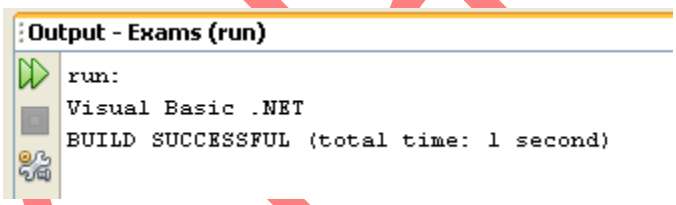
String exam = c1.examName("VB");

This is exactly the same as you did before: called the examName method of the StudentResults class. This time, however, you're using a Certificates object rather than a StudentResults object.

Add a print line and your code should look like this:

```
public class ExamDetails {

    public static void main(String[] args) {

//          StudentResults aStudent = new StudentResults();
//          String sName = aStudent.fullName("Bill Gates");
//          String exam = aStudent.examName("VB");
//          String score = aStudent.examScore(30);
//          String grade = aStudent.examGrade(30);
//          System.out.println( sName );
//          System.out.println( exam );
//          System.out.println( score );
//          System.out.println( grade );

            Certificates c1 = new Certificates();

            String exam = c1.examName("VB");
            System.out.println( exam );
    }
}
```

Run the programme to see the following Output window:



So the method from the parent class (the super class) has been called into action. We can now add a method to the child class (the sub class). Add the following method to your Certificates class, just below the constructor:

```
package exams;

public class Certificates extends StudentResults {

    private String certificate;

    Certificates() {
        super();
        certificate = "No Certificate Awarded";
    }

    String certificateAwarded(int aScore) {

        String aGrade = examGrade(aScore);

        if (aGrade.equals("Grade is A") ) {
            this.certificate = "Certificate of Excellence";
        }
        else if (aGrade.equals("Grade is B") ) {
            this.certificate = "Certificate of Achievement";
        }
        else if (aGrade.equals("Grade is C") ) {
            this.certificate = "Certificate of Achievement";
        }
        else  {
            this.certificate = "No Certificate Awarded";

        }

        return this.certificate;
    }
}
```

The method is called certificateAwarded, and is set up to return a String value. Inside the round brackets of the method, we're handing over an exam score.

The first line of the method is this:

String aGrade = examGrade(aScore);

The method examGrade is a method in the parent class. It's the one we set up in StudentResults. This method, remember, was set up to return a grade and some extra text, "Grade is A", "Grade is B", etc. We're now calling it from the child class. The IF Statement checks the value of the aGrade string to see what's in it. Depending on the value a new string is returned, awarding a particular certificate: Excellence, Achievement, or no certificate.

Click back on your ExamDetails class, and add the following line:

String award = c1.certificateAwarded(50);

This line calls the new method, and hands it a value of 50. The result is returned to the String we've called award.

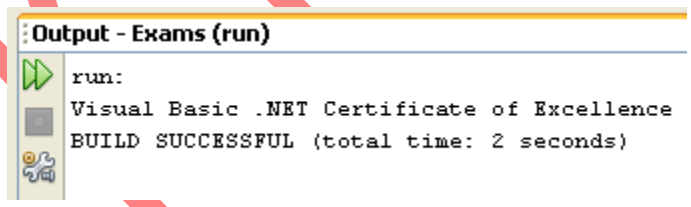
Adapt the print line method in your code to this:

System.out.println(exam + " " + award);

Your ExamDetails class should look like this (we've deleted all the comments):

```
package exams;

public class ExamDetails {

    public static void main(String[] args) {

            Certificates c1 = new Certificates();

            String exam = c1.examName("VB");
            String award = c1.certificateAwarded(50);

            System.out.println( exam  + " " + award );
    }
}
```

And here is the Output window when the programme is run:



So we've used Inheritance to keep the exam details separate from the certificate details. The child class (sub) accessed the methods of its parent (super). We were then able to output data from both.

Inheritance, then, is enhancing (extending) the usefulness of a parent class. You keep data separate by putting it in a child class. But the child is related in some way to the parent, and can access some or all of its code. Like children everywhere, however, it does its own thing!

Java Error Handling

In general, errors can be broken up into two categories: Design-time errors and Logical errors. Design-time errors are easy to spot because NetBeans usually underlines them. If the error will prevent the programme from running, NetBeans will underline it in red. Logical errors are the ones that you make as a programmer. The programme will run but, because you did something wrong with the coding, there's a good chance the entire thing will crash. You'll see examples of run-time errors shortly. You'll also see how to handle them. But first a word on how Java handles errors.

Exceptions

In Java, errors are handled by an **Exception** object. Exceptions are said to be thrown, and it's your job to catch them. You can do this with a try … catch block. The try … catch block looks like this:

```
try {

}

catch ( ExceptionType error_variable ) {

}
```

The **try** part of the **try … catch** block means "try this code". If something goes wrong, Java will jump to the catch block. It checks what you have between the round brackets to see if you have handled the error. If you have the correct Exception type then whatever code you have between the curly brackets of catch will get executed. If you don't have the correct Exception type then Java will use its default exception handler to display an error message.

As an example, create a new console application. Call it anything you like. In the code for the Main method, enter the following:

```
try {
int x = 10;
int y = 0;
int z = x / y;

System.out.println( z );
}
catch ( Exception err ) {
System.out.println( err.getMessage( ) );
}
```

In the **try** part of the **try … catch** block, we have set up three integers, x, y and z. We are trying to divide y into x, and then print out the answer.

If anything goes wrong, we have a catch part. In between the round brackets of catch we have this:

Exception err

The type of Exception you are using comes first. In this case we are using the Exception error object. This is a "catch all" type of Exception, and not very good programming practice. We'll change it to a specific type in a moment.

After your Exception type you have a space then a variable name. We've called ours **err**, but you can it almost anything you like.

In the curly brackets of catch we have a print statement. But look what we have between the round brackets of println:

err.getMessage( )

getMessage is a method available to Exception objects. As its name suggests, it gets the error message associated with the Exception.

Run your programme and test it out. Your code should look something like this:

```
package errorhandling;

public class errorChecking {

    public static void main(String[] args) {

        try {
            int x = 10;
            int y = 0;
            int z = x / y;

            System.out.println( z );
        }
        catch (Exception err) {
            System.out.println(err.getMessage());
        }
    }
}
```

And the Output window should display the following:

run:
/ by zero
BUILD SUCCESSFUL (total time: 1 second)

The error itself, the one generated by getMessage, is the line in the middle:

/ by zero

In other words, a divide by zero error. Java won't let you divide a number by zero, hence the error message.

Change your code to this:

```
double x = 10.0;
double y = 0.0;
double z = x / y;
```

The rest of the code can stay the same. Run your programme and test it out.

Again, an error message will be displayed in the Output window. This one:

```
run:
Infinity
BUILD SUCCESSFUL (total time: 1 second)
```

This time, Java stops the programme because the result will be an infinitely large number.

Errors that involve numbers shouldn't really be handled by a "catch all" Exception type. There is a specific type called ArithmeticException. Delete the word Exception between the round brackets of your catch block. Replace it with ArithmeticException. Now run the programme again.

You should find no difference in the error message displayed in the Output window. But you're using good programming practice by narrowing down the type of error you expect.

The Java Stack Trace

In the normal flow of a programme, when the Java Virtual Machine is running your code, one method after another will be executed, starting with the main method. When a method has its turn at the head of the programming queue it said be on top of the **stack**. After the whole of the method has been executed, it is taken off the stack to be replaced by the next method in the queue. To illustrate the principal, change your programme code to this:

```
package errorhandling;

public class errorChecking {

    public static void main(String[] args) {

        System.out.println("Starting Main method");
        m1();
        System.out.println("End Main method");
    }

    static void m1() {
        System.out.println("Method One - m1");
        m2();
    }

    static void m2() {
            System.out.println("Method Two - m2");
    }
}
```

We now have a Main method and two others: a method called **m1**, and a method called **m2**. When the programme first starts, the Main method is on top of the stack. However, inside of the Main method we have a call to the m1 method. When this method is called it jumps to the top of the stack. The m1 method in turn calls the m2 method. When m2 is called, it jumps to the top of the stack, pushing m1 aside temporarily. After m2 finishes, control is given back to m1. When m1 is finished, it gets pushed off the top of the stack, and control is handed back to the Main method.

Run your programme and watch the Output window to see what gets printed out:

```
Starting Main method
Method One - m1
Method Two - m2
End Main method
```

If something goes wrong in method m2, the JVM will look for any error handling, like a **try … catch** block. If there is no error handling, the Exception will get handed to m1 to see if it is dealing with the error. We don't have any error handling in m1 so again the Exception gets passed up the stack, this time to the Main method. If the Main method doesn't deal with the Exception then you'll get a strange error message printed to the Output window. As an example, adapt your **m2** method to this:

static void m2( ) {

int x = 10;
int y = 0;
double z = x / y;

System.out.println(z);
System.out.println("Method Two - m2");

}

The method contains the divide by zero error again. Your code should now look like ours below:

```
package errorhandling;

public class errorChecking {

    public static void main(String[] args) {

        System.out.println("Starting Main method");
        m1();
        System.out.println("End Main method");
    }

    static void m1() {
        System.out.println("Method One - m1");
        m2();
    }

    static void m2() {
            int x = 10;
            int y = 0;
            double z = x / y;

            System.out.println( z );
            System.out.println("Method Two - m2");
    }
}
```

Run the programme and watch what happens in the Output window:

```
Starting Main method
Method One - m1
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at errorhandling.errorChecking.m2(errorChecking.java:20)
        at errorhandling.errorChecking.m1(errorChecking.java:14)
        at errorhandling.errorChecking.main(errorChecking.java:8)
Java Result: 1
```

What you're looking at is something called a stack trace. The three lines in blue underline refer to your methods, and where they can be found:

package_name.class_name.method_name

The one at the top is where the error first occurred, in **m2**. Java was looking for this to be handled by an **ArithmeticException**, which is where divide by zero errors should be caught.

There was no error handling in m2, m1, or main. So the programme outputted to the default error handler.

Change you m1 method to this:

```
try {
System.out.println("Method One - m1");
m2( );
}
catch (ArithmeticException err) {
System.out.println( err.getMessage( ) );
}
```

We've now wrapped up the method called m2 in a **try** block. In the **catch** part, we've used the Exception type that was reported in the stack trace - ArithmeticException.

Run the code again, and the Output window will display the following:

```
Starting Main method
Method One - m1
/ by zero
End Main method
```

Notice that the error message is printed out: "/ by zero". The whole of the m2 method was not executed, but was stopped where the error occurred. Control was then passed back to m1. Because there was a catch block to handle the error, the JVM saw no need to a default error handler, but printed out the message between the curly brackets of catch.

The programme itself was not stopped, however. Control was passed back to the Main method, where the m1 method was called. The final line in the Main method, printing out "End Main method", was executed. This has important implications. Suppose you needed the value from m1, because you were going to do something with it in Main. The value wouldn't be there, and your programme may not behave as expected.

But if you see a stack trace in the Output window, just remember that first line is where the problem occurred; the rest of the lines, if any, are where the Exception was handed up the stack, usually finishing at the main method.

Logic Errors in Java

Logic errors are the ones you make as a programmer, when the code doesn't work as you expected it to. These can be hard to track down. Fortunately, NetBeans has some built-in tools to help you locate the problem.

First, examine this code:

```
package errorhandling2;

public class errorChecking2 {

    public static void main(String[] args) {
        int LetterCount = 0;
        String check_word = "Debugging";
        String single_letter = "";
        int i;

        for (i = 0; i < check_word.length(); i++) {

            single_letter = check_word.substring(1, 1);

            if (single_letter.equals("g")) {
                LetterCount++;
            }
        }

        System.out.println("G was found " + LetterCount + " times.");
    }
}
```

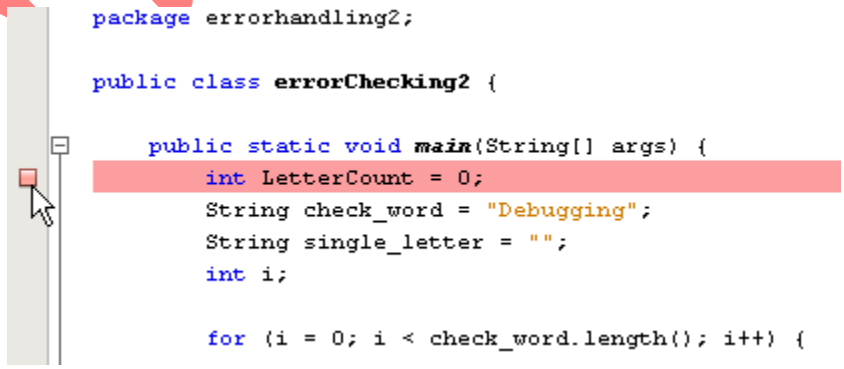
Type it out for yourself, using either the programme you already have, or by starting a new project. What we're trying to do here is count how many times the letter "g" occurs in the word "Debugging". The answer is obviously 3. However, when you run the programme the Output window prints:

"G was found 0 times."

So we've made an error somewhere in our code. But where is it? The programme executes OK, and doesn't throw up any Exceptions for us to study in the Output window. So what to do?

To track down problems with your code, NetBeans allows you to add something called a Breakpoint.

Click in the margins of the code window to add a new Breakpoint:



From the NetBeans menu, click **Debug > Debug** errorhandling2 (or whatever you called your project). NetBeans will jump to the breakpoint. It's has now halted code execution at that point. You should also see a new toolbar appear:

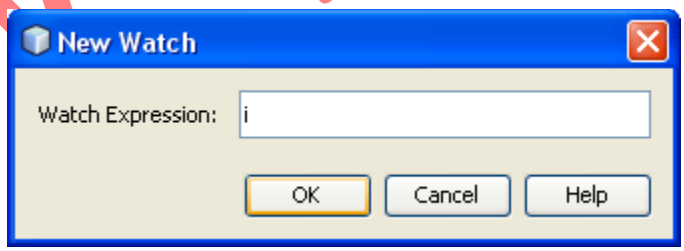
The first three buttons allow you to stop the debugging session, pause, and continue. The next five buttons allow you to step into code, step over code, step out, or jump to the cursor.

You can also press the F5 key to continue. The code should run as normal with the Breakpoint where it is. The debugging session will then end.

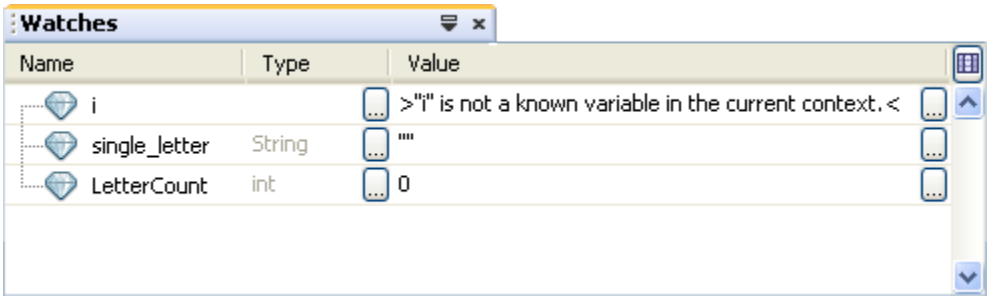
When the debugging session ends, click back on the Breakpoint to get rid of it. Now add a Breakpoint to your **for** loop:

```
public class errorChecking2 {

    public static void main(String[] args) {
        int LetterCount = 0;
        String check_word = "Debugging";
        String single_letter = "";
        int i;

        for (i = 0; i < check_word.length(); i++) {

            single_letter = check_word.substring(1, 1);
```

Now click **Debug > New Watch**. A Watch allows you to keep track of what's in a variable. So type the letter **i** in the Watch dialogue box and click OK:

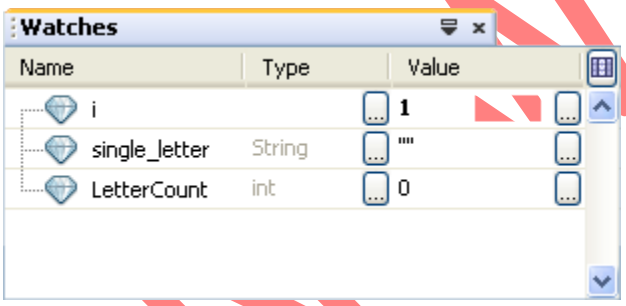


Add another Watch and then type **single_letter**. Click OK. Add a third Watch and type **LetterCount**. You should these three variables at the bottom of your screen:



Now press the Step Into icon on the toolbar:

Or just press the F7 key on your keyboard. Keep pressing the F7 key and see what happens in the Watch window. You should find that the i variable goes up by 1 each time. But the other two variables stay the same:



Because there's nothing in the single_letter variable then LetterCount can't go beyond zero. So we've found our problem - our use of substring must be wrong, as it's not grabbing any characters.

Stop the debugging session and change your substring line to this:

single_letter = check_word.substring(i, i + 1);

Now start the debugging session again. Keep pressing the F7 key to go over each line of the for loop. This time you should see the sinle_letter and LetterCount variables change.

When the code ends, you should see the Output window display the following:

"G was found 3 times."

We now have the correct answer.

So if things are not going as planned with your code, try setting a Breakpoint and some Watches for your variables. Then start a debugging session.

How to Read a Text File in Java

Manipulating text files is a skill that will serve you well in your programming career. In this section, you'll learn how to open and how to write to a text file. But by text file, we just mean a file with text in it - simple as that! You can create a text file in programmes like Notepad on a Windows computer, TextEdit on a Mac, Gedit in a Linux/Gnome environment.

The first thing we'll do is to open up a text file and read its contents.

Read a Text File

Start a new project for this. Call the package **textfiles** and the class **FileData**. Add an import statement just below the package line and before the class name:

import java.io.IOException;

Your coding window will then look like this:

```
package textfiles;
import java.io.IOException;

public class FileData {

    public static void main(String[] args)  {

    }

}
```

To deal with anything going wrong with our file handling, add the following to the main method (the text in bold):

public static void main(String[ ] args) throws IOException {

}

We're telling Java that the main method will throw up an **IOException** error, and that it has to be dealt with. Later, we'll add a **try … catch** block to display an appropriate error message for the user, should something go wrong.

To open the text file, let's create a new class. So click **File > New File** from the NetBeans menu at the top. Create a new Java Class file and give it the name **ReadFile**. When your new class is created, add the following three import statements:

import java.io.IOException;
import java.io.FileReader;
import java.io.BufferedReader;

Your new class should then look like this one:

```
package textfiles;

import java.io.IOException;
import java.io.FileReader;
import java.io.BufferedReader;

public class ReadFile {

}
```

(The import lines are underlined because we haven't done anything with them yet. This is a NetBeans feature.)

We'll create a new object from this class to read a file. Add the following constructor to your code, along with the private String field called **path**:

```
public class ReadFile {

    private String path;

    public ReadFile(String file_path) {
        path = file_path;
    }
}
```

All we're doing here is passing in the name of a file, and then handing the file name over to the path field.

What we now need to do is create a method that returns all the lines of code from the text file. The lines will be held in an array. Add the following method declaration that will open up the file:

```
public class ReadFile {

    private String path;

    public ReadFile(String file_path) {
        path = file_path;
    }

    public String[] OpenFile() throws IOException {


    }
}
```

Don't worry about the red underline: it will go away once we've added some code. NetBeans has just added it because we have no return statement.

Notice that the method is set up to return a String array, though:

public String[ ]

The array will contain all the lines from the text file.

Notice, too, that we've added "throws IOException" to the end of the method header. Every method that deals with reading text files needs one of these. Java will throw any errors up the line, and they will be caught in our main method.

To read characters from a text file, the **FileReader** is used. This reads bytes from a text file, and each byte is a single character. You can read whole lines of text, rather than single characters. To do this, you can hand your FileReader over to something called a **BufferedReader**. The BufferedReader has a handy method called ReadLine. As its name suggests, it is used to read whole lines, rather than single characters. What the BufferedReader does, though, is to store characters in memory (the buffer) so that they can be manipulated more easily.

Add the following lines that set up a FileReader and a BufferedReader:

```
public String[] OpenFile() throws IOException {

    FileReader fr = new FileReader(path);
    BufferedReader textReader = new BufferedReader(fr);

}
```

We're creating two new objects here: one is a FileReader object which we've called **fr**; the other is a BufferedReader object with the name **textReader**.

The FileReader needs the name of file to open. For us, the file path and name is held in the field variable called path. So we can use this.

The BufferedReader is handed the FileReader object between its round brackets. All the characters from the file are then held in memory waiting to be manipulated. They are held under the variable name **textReader**.

Before we can read the lines of text, we need to set up an array. Each position in the array can then hold one complete line of text. So add the following two lines to your code:

```
int numberOfLines = 3;
String[ ] textData = new String[numberOfLines];
```

For now, we'll set the number of lines in the text file to just 3. Obviously, text files can hold any number of lines, and we usually don't know how many. So we'll change this soon. We'll write a separate method that gets the number of lines in a text file.

The second line of new code, though, sets up a String array. The number of positions in the array (its size) is set to the number of lines. We've put this between the square brackets.

To put all the lines of text from the file into each position in the array, we need a loop. The loop will get each line of text and place each line into the array. Add the following to your code:

```
int i;

for (i=0; i < numberOfLines; i++) {
textData[ i ] = textReader.readLine();

}
```

Your coding window should now look like this:

```
public String[] OpenFile() throws IOException {

    FileReader fr = new FileReader(path);
    BufferedReader textReader = new BufferedReader(fr);

    int numberOfLines = 3;
    String[] textData = new String[numberOfLines];

    int i;

    for (i=0; i < numberOfLines; i++) {
        textData[i] = textReader.readLine();
    }

}
```

The for loop goes from 0 to just less than the number of lines. (Array positions, remember, start at 0. The 3 lines will be stored at positions 0, 1, and 2.)

The line that accesses the lines of text and stores them in the array is this one:

textData[i] = textReader.readLine( );

After the equals sign we have this:

textReader.readLine( );

The textReader object we set up is holding all the characters from the text file in memory (the buffer). We can use the readLine method to read a complete line from the buffer. After the line is read, we store the line in an array position:

textData[i]

The variable called **i** will increment each time round the loop, thus going through the entire array storing lines of text.

Only two more lines of code to add to the method, now. So add these lines to your code:

textReader.close( );
return textData;

The close method flushes the temporary memory buffer called textReader. The return line returns the whole array. Notice that no square brackets are needed for the array name.

When you've added the code, all those ugly underlines should disappear. Your method should then look like this:

```
public String[] OpenFile() throws IOException {

    FileReader fr = new FileReader(path);
    BufferedReader textReader = new BufferedReader(fr);

    int numberOfLines = 3;
    String[] textData = new String[numberOfLines];

    int i;

    for (i=0; i < numberOfLines; i++) {
        textData[i] = textReader.readLine();
    }

    textReader.close();
    return textData;
}
```

There's still the problem of the number of lines, however. We've hard-coded this to 3. What we need is to go through any text file and count how many lines it has. So add the following method to your ReadFile class:

```
int readLines() throws IOException {

   FileReader file_to_read = new FileReader(path);
   BufferedReader bf = new BufferedReader(file_to_read);

   String aLine;
   int numberOfLines = 0;

   while (( aLine = bf.readLine()) != null) {
           numberOfLines++;
   }
   bf.close();

   return numberOfLines;

}
```

The new method is called **readLines**, and is set to return an integer value. This is the number of lines a text file has. Notice this method also has an **IOException** part to the method header.

The code for the method sets up another FileReader, and another BufferedReader. To loop round the lines of text, we have this:

```
while ( ( aLine = bf.readLine( ) ) != null ) {
numberOfLines++;
}
```

The while loop looks a bit messy. But it just says "read each line of text and stop when a null value is reached." (If there's no more lines in a text file, Java returns a value of null.) Inside the curly brackets, we increment a counter called numberOfLines.

The final two lines of code flush the memory buffer called bf, and returns the number of lines.

To call this new method into action, change this line in your OpenFile method:

int numberOfLines = 3;

Change it to this:

int numberOfLines = readLines( );

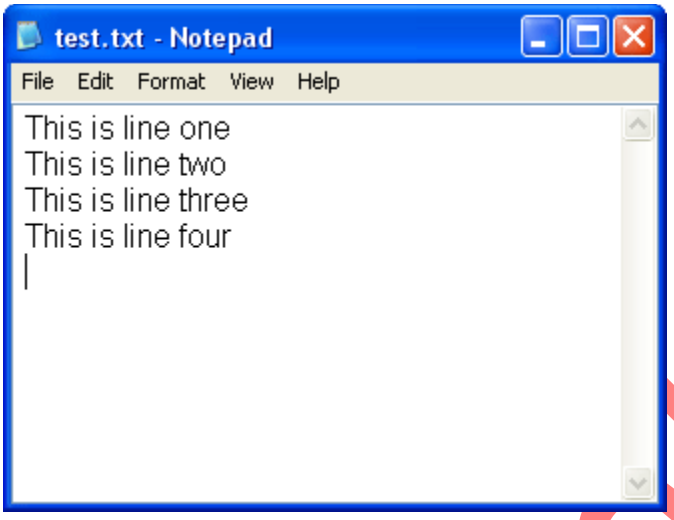
So instead of hard-coding the number of lines, we can call our new method and get the number of lines in any text file.

OK, time to put the new class to work and see if it opens a text file.

Go back to your **FileData** class, the one with the main method in it. Set up a string variable to hold the name of a text file:

```
package textfiles;
import java.io.IOException;

public class FileData {

    public static void main(String[] args) throws IOException {

        String file_name = "C:/test.txt";

    }
}
```

At this stage, you need to create a text file somewhere on your computer. We created this simple one in Notepad on a Windows machine:



The name of the text file is "test.txt". Create a similar text file on your own computer. Note where you saved it to because you need the file path as well:

String file_name = "C:/test.txt";

So our **test.txt** file is saved on the C drive. If we had created a folder called MyFiles to hold the file then the path would be "C:/MyFiles/test.txt". Change you file path, if need be.

The next thing to do is to create a new object from our **ReadFile** class. We can then call the method that opens the file. But we can do this in a **try … catch** block. Add the following code, just below your String variable line:

```
public static void main(String[] args) throws IOException {

    String file_name = "C:/test.txt";

    try {
        ReadFile file = new ReadFile(file_name);
        String[] aryLines = file.OpenFile();

    }
    catch (IOException e) {
          System.out.println( e.getMessage() );
    }
}
```

Don't forget all the curly brackets for the try … catch block. You need one pair for the try part and another pair for the catch part. For the try part, we have this:

ReadFile file = new ReadFile(file_name);
String[ ] aryLines = file.OpenFile( );

The first line sets up a new ReadFile object called **file**. In between the round brackets of ReadFile, we added the **file_name** variable. This is enough to hand the constructor the file path it needs.

The second line of code sets up a String array called **aryLines**. After the equals sign, we've called the **OpenFile** method of our ReadFile class. If it successfully opens up the text file, then the array of text lines will be handed over to the new array aryLines.

If something goes wrong, however, an error is thrown up the line, and ends up in the catch part of the **try … catch** block:

```
catch ( IOException e ) {
System.out.println( e.getMessage() );
}
```

After the word "catch" we have a pair of round brackets. Inside the round brackets, we have this:

IOException e

What this does is to set up a variable called e which is of type **IOException**. The IOException object has methods of its own that you can use. One of these methods is getMessage. The will give the user some information on what went wrong.

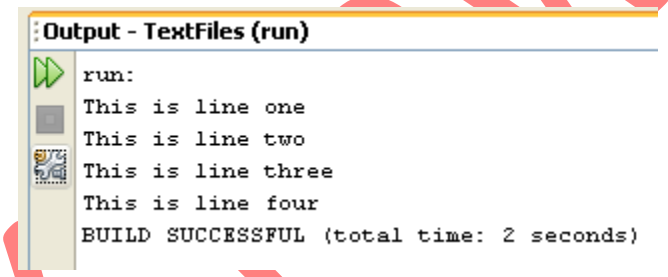
Before we see an example of an error message, let's loop through all the lines of the text file, printing out each one. Add the following loop code to the **try** part of the **try … catch** block:

```
int i;
for ( i=0; i < aryLines.length; i++ ) {
System.out.println( aryLines[ i ] ) ;
}
```

Your coding window should now look like this:

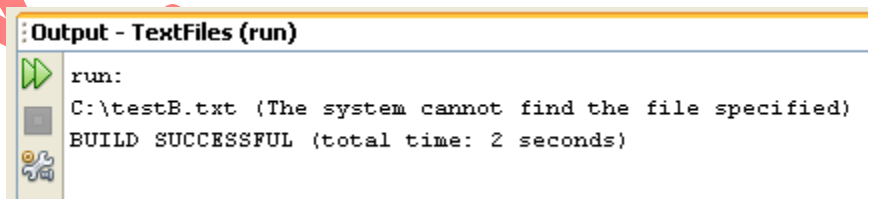
```
public static void main(String[] args) throws IOException {

    String file_name = "C:/test.txt";

    try {
        ReadFile file = new ReadFile(file_name);
        String[] aryLines = file.OpenFile();

        int i;
        for (i=0; i < aryLines.length; i++) {
            System.out.println(aryLines[i]);
        }
    }
    catch (IOException e) {
          System.out.println( e.getMessage() );
    }
}
```

When the programme is run, the Output window will print the following:



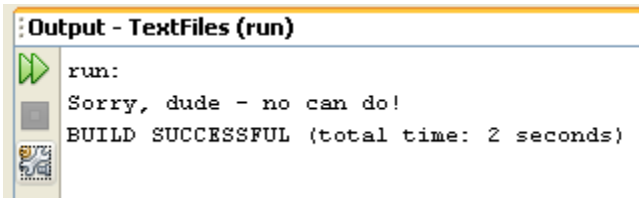
As you can see, each line from our text file has been printed out.

To test the error checking part of the code, change the name of your text file to one you know has not been created. Then run your code again. In the Output window below, you can see that our text file was changed to **testB**, and that it couldn't be found:



If you prefer, you can add your own error message to the catch block:

```
catch (IOException e) {
      System.out.println( "Sorry, dude - no can do!" );
}
```



It's probably better to leave it to Java, though!

Write to a Text File in Java

Writing to a file is a little easier than reading a file. To write to a file, we'll use two more inbuilt classes: the **FileWriter** class and the **PrintWriter** class.

Create a new class in your project by clicking **File > New File** from the NetBeans menu. Select **Java** in the Categories section of the dialogue box and **Class** from the File Types list. Click the Next button at the bottom. For the Class name type **WriteFile**, then click Finish. Add the following three import statements your new code:

import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;

Your new class should look like this:

```
package textfiles;

import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;

public class WriteFile {


}
```

Again, the underlines are because we haven't used the imported classes yet.

When you write to a file, you can either start from the beginning and overwrite everything. Or you can start at the end and append to the file. The FileWriter class allows you to specify which. We'll add a field that sets the append value for the FileWriter class. We'll also add a field to set the name of the file.

So add the following two fields to your code, plus the constructor:

```
public class WriteFile {

    private String path;
    private boolean append_to_file = false;

    public WriteFile(String file_path) {
        path = file_path;
    }
}
```

The boolean field is called **append_to_file** and has been set to a value of **false**. This is the default value for the FileWriter class, and means you don't want to append, but erase everything in the file.

The constructor sets a value for the **path** field (instance variable), which is the name and location of the file. This will get handed over when we create a new object from our WriteFile class.

As was mentioned in the previous section, however, you can set up more than one constructor in your code. We can set up a second one and pass in an append value. That way, a user can either just use the first constructor and hand over a file name, or a file name and an append value. So add the following constructor below the first one:

public WriteFile(String file_path , boolean append_value) {

path = file_path;
append_to_file = append_value;

}

This second constructor now has two values between the round brackets, a file path and an append value. If you want to append to the file you can use this constructor when creating a new object. If you just want to overwrite the text file then you can use the first constructor.

Your code window should now look like this:

```
package textfiles;

import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;

public class WriteFile {

 private String path;
 private boolean append_to_file = false;

 public WriteFile(String file_path) {
     path = file_path;
 }

 public WriteFile( String file_path, boolean append_value ){
     path = file_path;
     append_to_file = append_value;
 }

}
```

To write to the file, add the following method below your two constructors:

public void writeToFile(String textLine) throws IOException {

}

This method doesn't need to return a value, so we've made it void. In between the round brackets of the method name we have a String variable called **textLine**. This is obviously the text we want to write to the file. Again, though, we need to add "throws IOException" as we need to do something to handle file-writing errors.

The first thing we need in the method is a FileWriter object. The FileWriter takes care of opening the correct file, and of storing the text as bytes. Add the following line to your writeToFile method:

FileWriter write = new FileWriter(path , append_to_file);

So we're creating a new FileWriter object with the name write. In between the round brackets of FileWriter we pass the name and location of the file, plus the append value. This will either be **true** (append to the file) or **false** (don't append). If a file of the name you pass over does not exist, the FileWriter creates one for you.

The FileWriter write bytes, however. But we can hand the FileWriter plain text with the aid of the **PrintWriter** class. The PrintWriter has a few handy print methods for this. But it needs the name of a FileWriter when creating the object from the class. So add this line to your method:

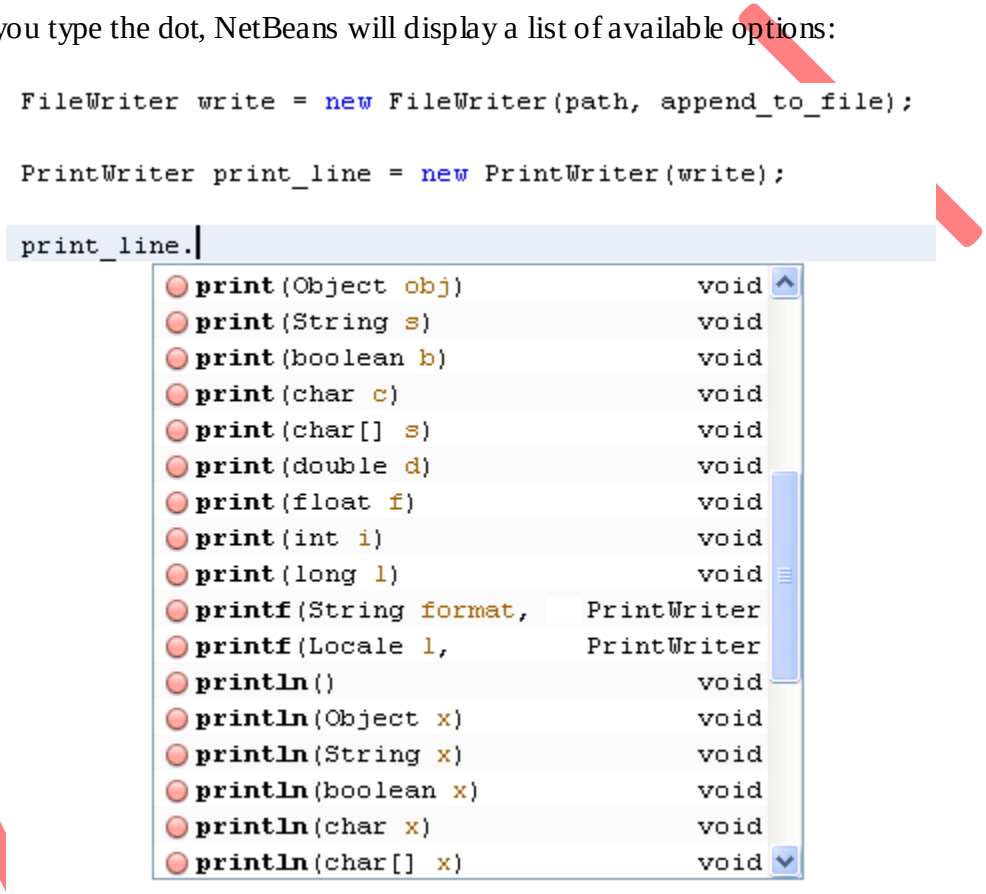
PrintWriter print_line = new PrintWriter(write);

Our PrintWriter object is called print_line. In between the round brackets of PrintWriter, we've added the name of our FileWriter object.

To add the text to a file, type the name of the PrintWriter object followed by a dot:

print_line.

As soon as you type the dot, NetBeans will display a list of available options:



There are an awful lot of print options on the list!

The one we'll use is one of the **printf** methods. This allows you to pass a formatted string of text to your PrintWriter. A good reason for using printf is to handle new line characters. The new line character differs, depending on which operating system you use. Windows will add the characters \r\n for a new line. But Unix systems just use \n. Using the printf function will ensure the correct encoding, no matter what the platform.

Add the following line to your code:

print_line.printf("%s" + "%n" , textLine);

We've handed the printf method two things: a format for the text, and the string we want to write to the file. The two are separated by a comma. Notice the first of the two:

"%s" + "%n"

The %s between double quotes means a string of characters of any length. The %n means a newline. So we're telling the printf method to format a string of characters and add a newline at the end. The actual text that needs formatting goes after the comma. The printf method is quite useful, and we'll go through the options in more detail in a later section. For now, let's crack on.

Only one more line to add to your method:

print_line.close();

This line closes the text file and frees up any resources it was using.

Your WriteFile class should now look like this:

```
package textfiles;

import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;

public class WriteFile {

 private String path;
 private boolean append_to_file = false;

 public WriteFile(String file_path) {
     path = file_path;
 }

 public WriteFile( String file_path, boolean append_value ) {
     path = file_path;
     append_to_file = append_value;
 }

public void writeToFile(String textLine) throws IOException {

   FileWriter write = new FileWriter(path, append_to_file);
   PrintWriter print_line = new PrintWriter(write);

   print_line.printf("%s" + "%n", textLine);

   print_line.close();

 }
}
```

To test out your new class, go back to your FileData class (the one with the main method). Add the following line to create a new object from your WriteFile class:

WriteFile data = new WriteFile(file_name , true);

So we've set up a WriteFile object called data. In between the round brackets of WriteFile, we've added two things: the name of the file, and an append value of true. This will ensure that the second of the constructors we set up gets called. If we wanted to just overwrite the file, then the code would be this:

WriteFile data = new WriteFile(file_name);

Because we set the default append value as false, we only need the file name if we want to overwrite the entire contents.

To call the writeToFile method of your WriteFile object, add this line:

data.writeToFile("This is another line of text");

Feel free to change the text between the round brackets of the method.

To let the user know that something has happened, you can print something to the Output window:

System.out.println("Text File Written To");

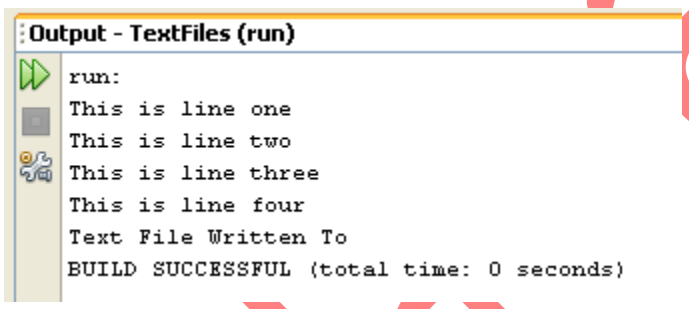
Your **FileData** code should now look like this (we've added some comments):

```
package textfiles;
import java.io.IOException;

public class FileData {

    public static void main(String[] args) throws IOException {
        String file_name = "C:/test.txt";
        try {
            ReadFile file = new ReadFile(file_name);
            String[] aryLines = file.OpenFile();
            int i;
            for (i=0; i < aryLines.length; i++) {
                System.out.println(aryLines[i]);
            }
        }
        catch (IOException e) {
              System.out.println( "Sorry, dude - no can do!" );
        }

        //============================
        //  WRITE TO A FILE
        //============================
        WriteFile data = new WriteFile(file_name, true);
        data.writeToFile("This is another line of text");

        System.out.println("Text File Written To");
    }
}
```

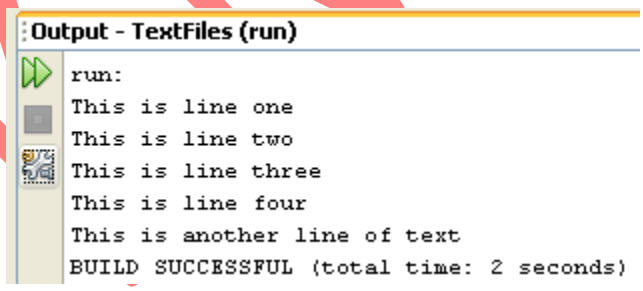
If you like, add another **try … catch** part for your text writing. Instead of the above, change it to this:

```
//============================
//  WRITE TO A FILE
//============================
try {
   WriteFile data = new WriteFile(file_name, true);
   data.writeToFile("This is another line of text");
}
catch (IOException e) {
      System.out.println( e.getMessage() );
}
```

Now run your code to test it out. You should see the contents of your text file in the Output window followed by the message that the text file has been written to:



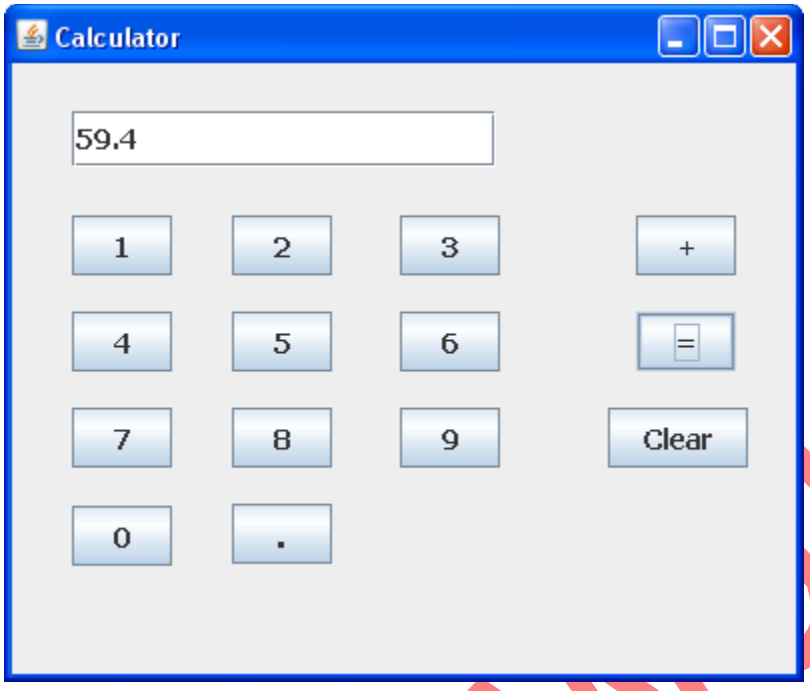
Run the programme again and you should see the new line appear. (You can comment out the code that writes to the text file.)



And that's it - you can now write to a text file and read its contents.

Creating a Java Form with the NetBeans IDE

You don't have to output everything to a terminal window in Java. In this section, you'll be writing a calculator programme that makes use of forms. The form will have buttons and a text box. We'll start off with a very simple calculator that can only add up, and then extend its capabilities so that it can subtract, divide and multiply. The calculator will look something like this:



(The above screenshot was taken on a Windows XP machine. Your calculator will look different if you have, say, a Linux operating system, or an Apple Mac.)

Let's make a start.

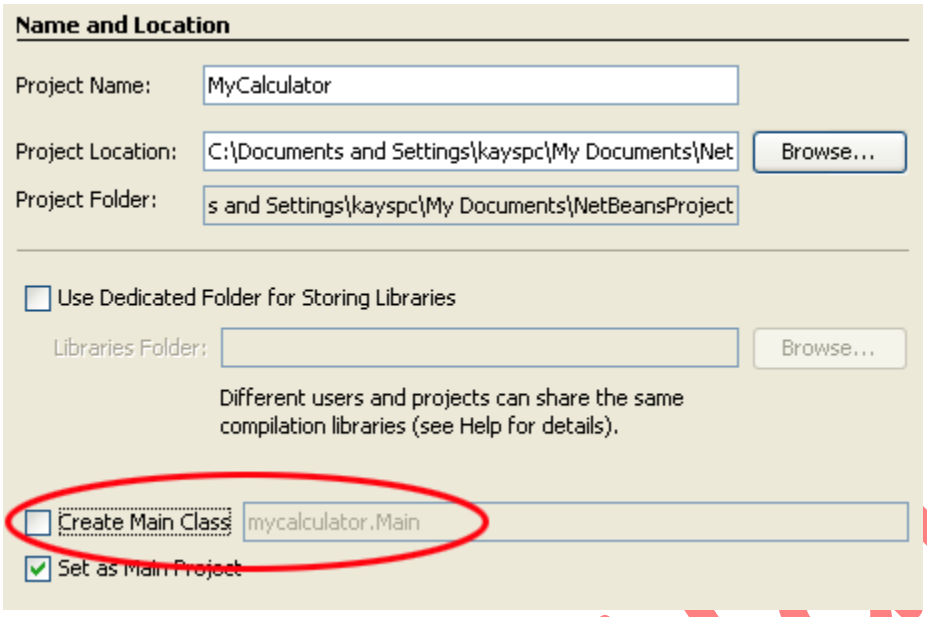
NetBeans and Java Forms

Developing a Graphic User Interface (GUI) using Java can be an art-form in itself, as there's quite a lot to get used to: Components, Containers, Layout Managers, and a whole lot more besides. The NetBeans development environment, however, has greatly simplified the creation of forms, and we'll use this to drag and drop controls onto a frame.

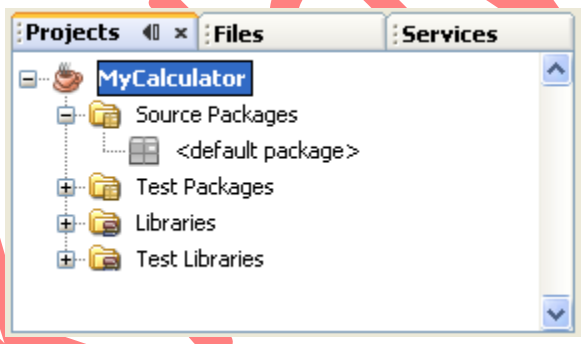
Rather than go through reams of GUI theory, we'll just jump straight into it.

Create a new project for this by clicking **File > New Project** from the NetBeans menu at the top. Select **Java > Java Application** from the list boxes, and then click the Next button.

On step 2 of the wizard, type **MyCalculator** as the project name. At the bottom, uncheck "Create main class". This is because a main method will be created for us by NetBeans when we add a form. Step 2 of the wizard should look like this:



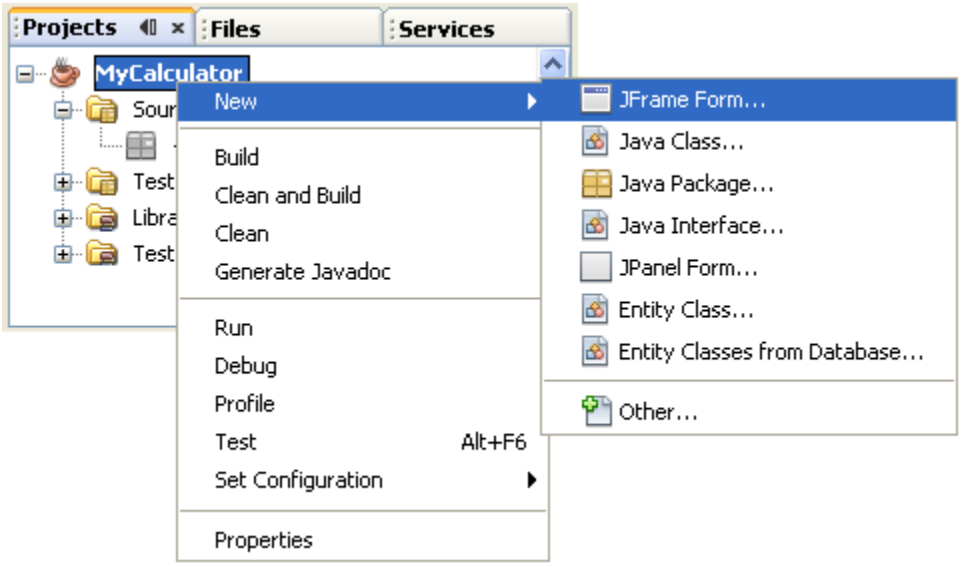
Click the Finish button, and NetBeans will create the project, but not much else. Have a look at the Projects area on the left in NetBeans and you should see this (If you can't see the Projects area, click **Window > Projects** from the NetBeans menu at the top):



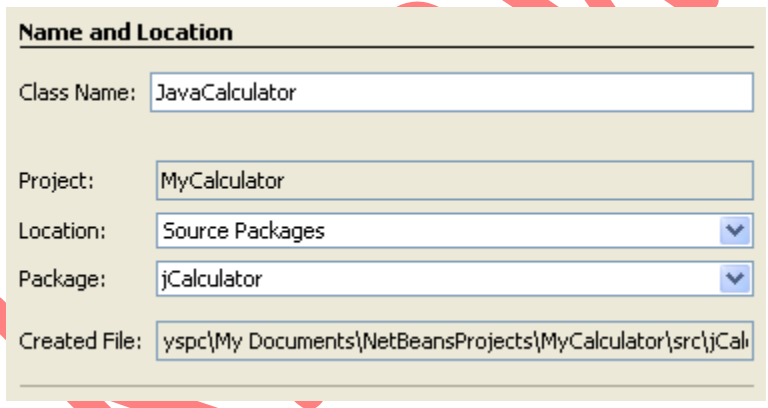
Normally, there's a .**java** file under the Source Packages name. But because we unchecked the "Create main class" box, there's no java class file there.

What we'll do is to add a Form to the project. When the form is created, it will be created in its own java class file.

To add a form, right click the project name in the Projects window. A menu will appear:



Select **New > JFrame** Form from the menu. When you do, you should see the following wizard appear:

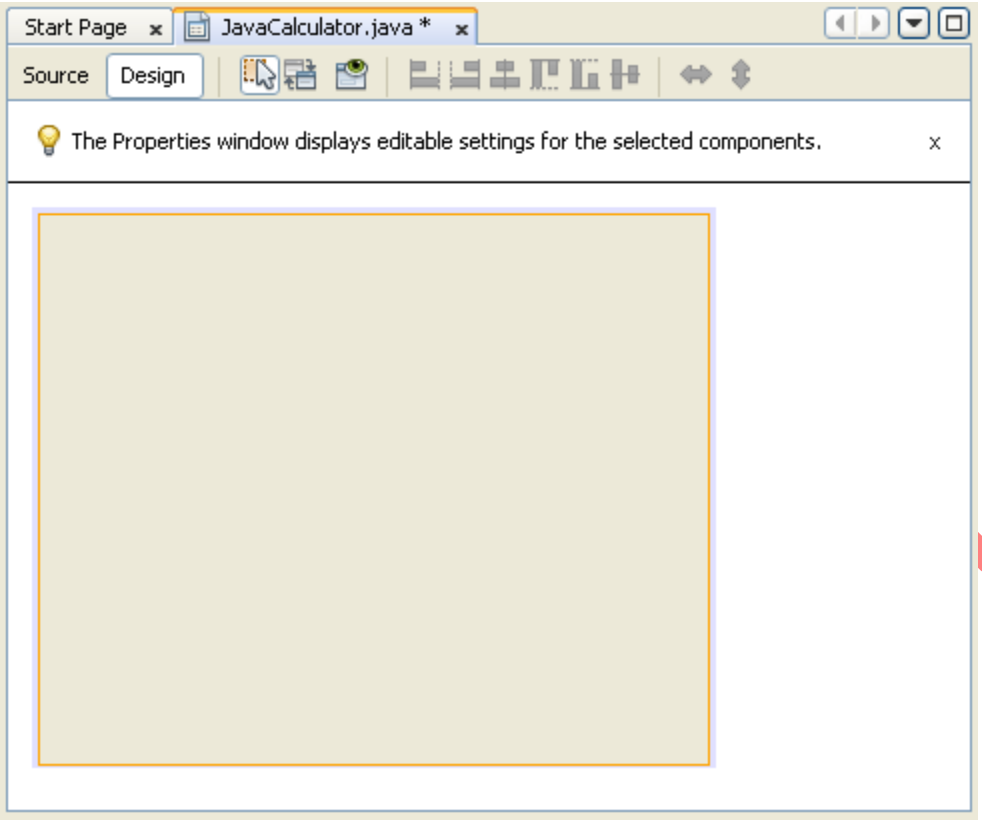


Here, you are being asked for a name for your Class, and a package name. We've already created the project, and called it **MyCalculator**. The package name and class will go into the project. So, for your Class Name type **JavaCalculator**. In the blank package text box, type **jCalculator**. So we're creating a class called JavaCalculator, which is in the jCalculator package, which is in the MyCalculator project.

Click the Finish button to complete the process.

Form Views

When the wizard from the previous section has finished, it will create a blank form in the main window:



The form is a blank, at the moment, and has an orange rectangle surrounding it. The orange rectangle means that the form is the currently selected object. Click away and you'll see a blue rectangle instead. This means that the form is not selected. Click back onto the form to select it, and you'll see the orange rectangle again.

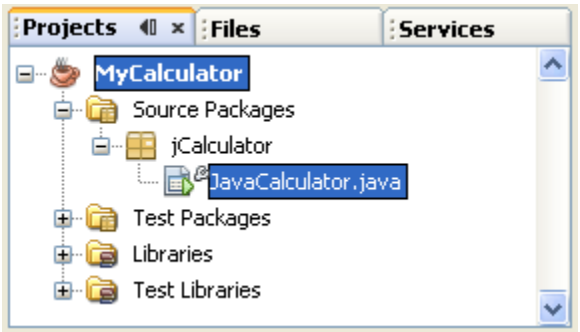
Note the two buttons at the top, **Source** and **Design**. You're in Design at the moment. Click on Source to see the following code:

```
package jCalculator;

/**...*/
public class JavaCalculator extends javax.swing.JFrame {

    /** Creates new form JavaCalculator */
    public JavaCalculator() {
        initComponents();
    }

    /**...*/
    @SuppressWarnings("unchecked")
    Generated Code

    /**...*/
    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new JavaCalculator().setVisible(true);
            }
        });
    }

    // Variables declaration - do not modify
    // End of variables declaration

}
```

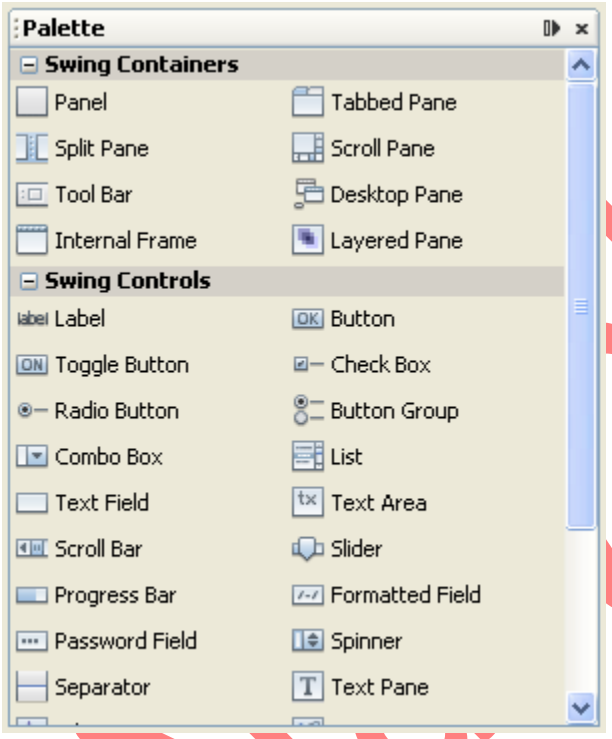
You can expand and contract the plus symbols to reveal and hide code. Notice, though, that a **main** method has been created. When the programme starts it's the main method that will be called. It's creating a new object from our form, and setting its visible property to **true**.

But NetBeans generates all this code for you. Otherwise, you'd have to do it all yourself!

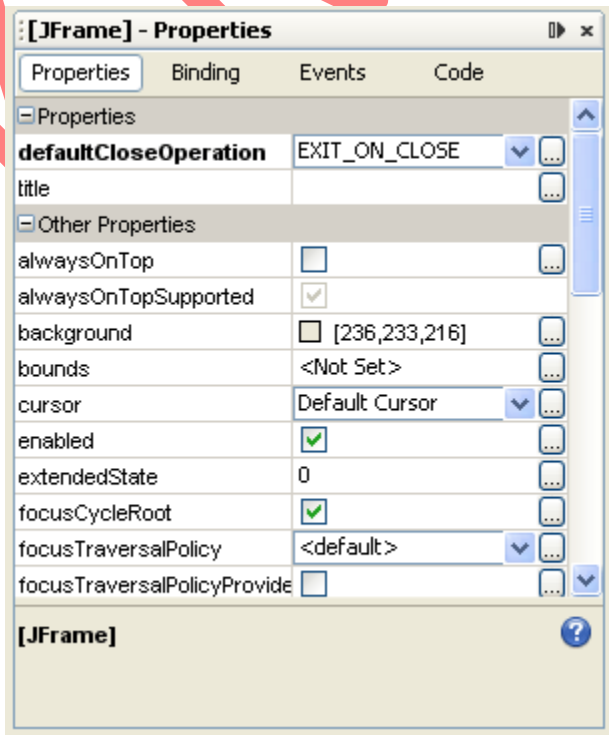
Have a look at the Projects area on the left again. You'll see that a package and a class file have been added:



Click back on the Design button at the top. You'll see your blank form again. To the right, you'll have noticed two areas: a Palette with a lot of controls in it, and a Properties area. The Palette should look like this:



And the Properties area should look like this:

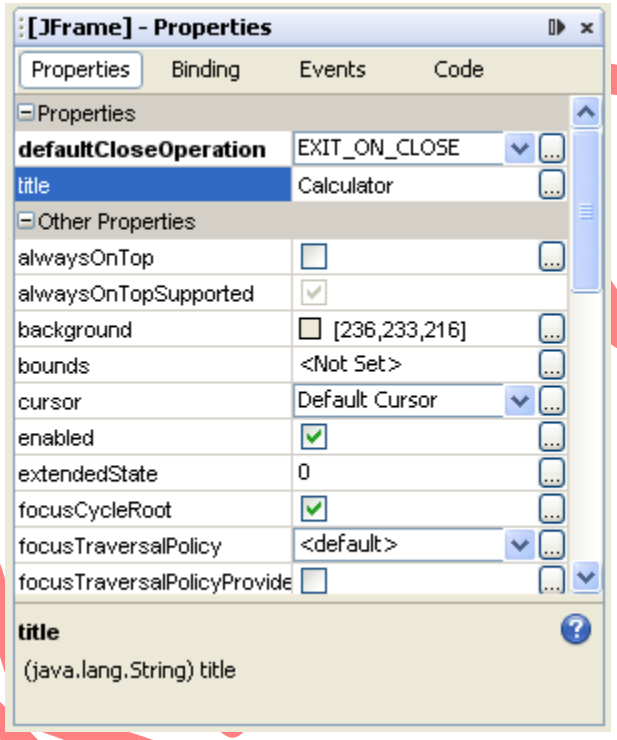


(If you can't see them, click **Window> Palette** and **Window > Properties** from the NetBeans menu.)

A property of an object is a list of the things you can do with it, like set the background colour, set the text, set the font, and lots more. Let's change the title.

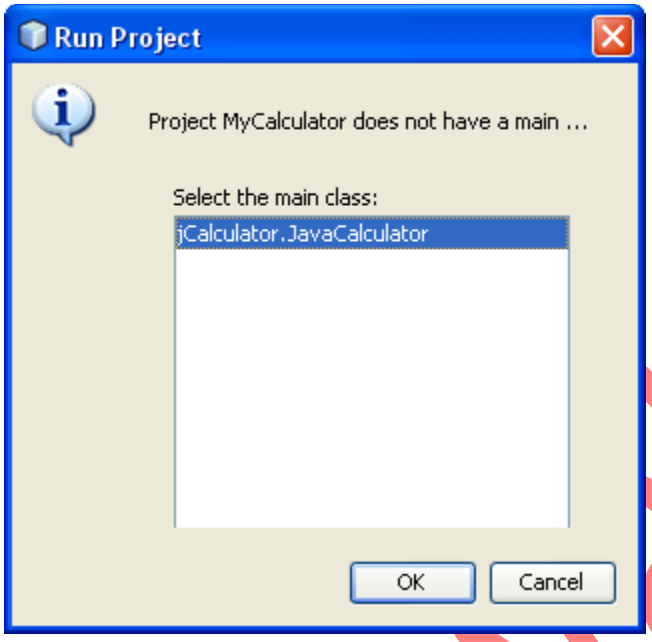
Make sure your form is selected. If it is, it will be surrounded with an orange rectangle. The properties area will say JFrame at the top.

Click inside of the **title** area and type Calculator:

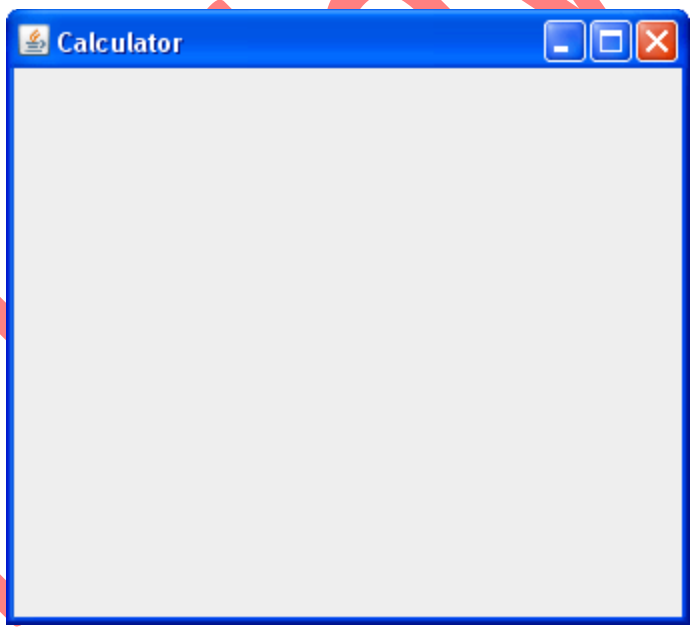


Then press the enter key.

To see what effect changing the title property has, run your programme in the usual way. When you run the programme for the first time, NetBeans will display a dialogue box asking you which main method you want to use:



You only have one, which is in your JavaCalculator class, so just click OK. Your form should then appear:

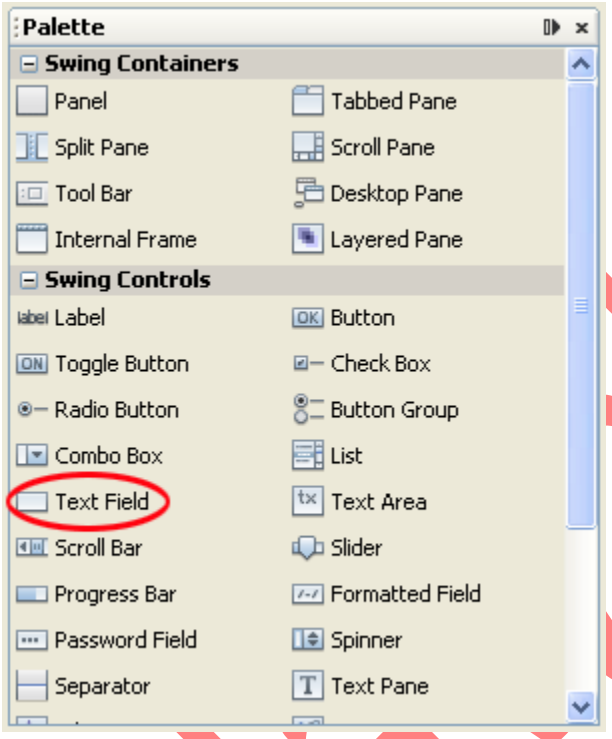


The form has the title you just typed, plus some default icons for minimize, maximize and close. Click the X to close your programme, and return to the Design environment.

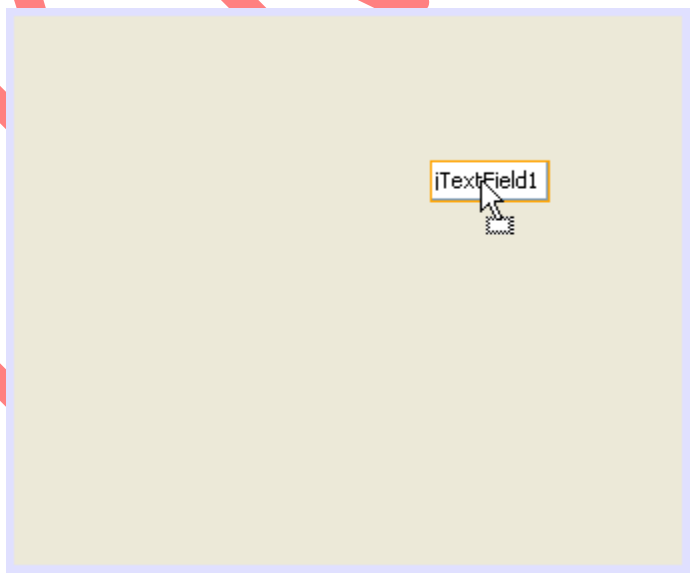
Add a Text Box to a Java Form

What our form needs now is a text box and some buttons. Let's add the text box first.

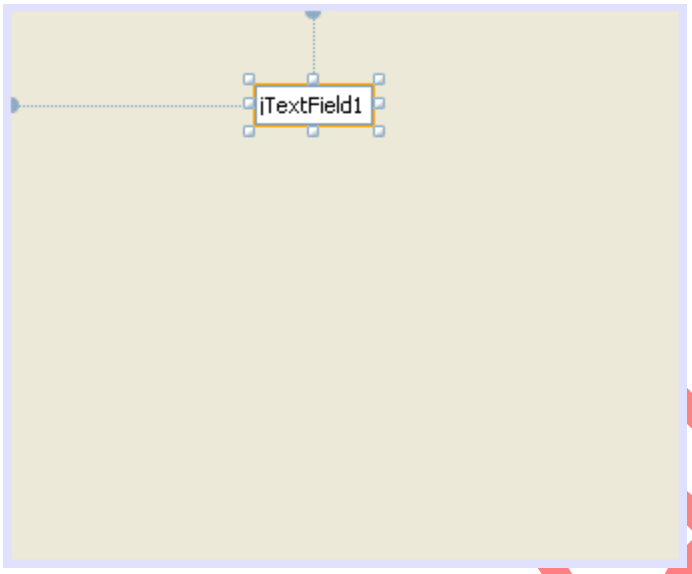
Locate the Text Field control in the Palette:



Controls in the NetBeans Palette can be dragged onto a form. So click on Text Field to select it. Now hold you left mouse button down on the Text Field. Keep it held down and drag the control onto the form:

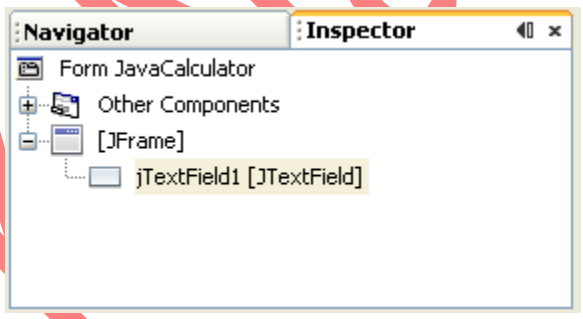


Let go anywhere on the form:

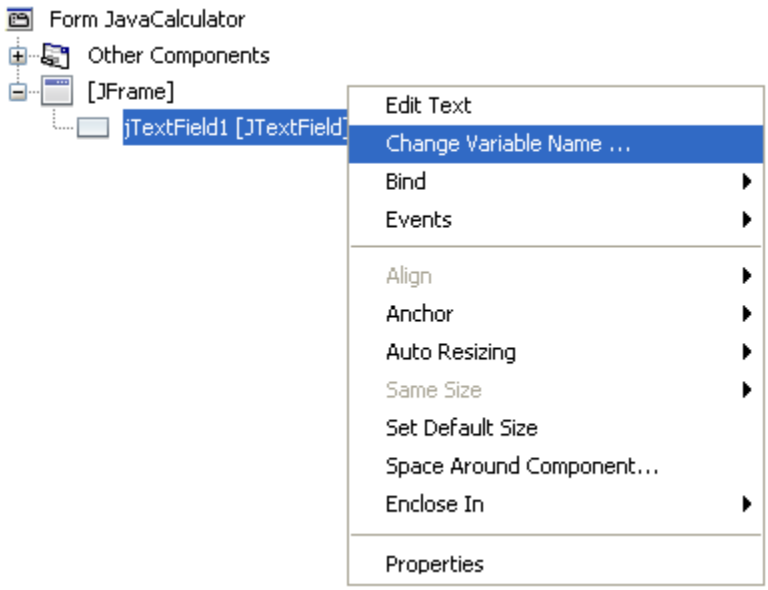


The squares around the text field are sizing handles. You can hold down your left mouse button on one of the squares and drag to a new width or height. The dotted lines are position indicators, one for the left position and one for the top position. Notice that the default name for the text field is **jTextField1**. Let's change that.

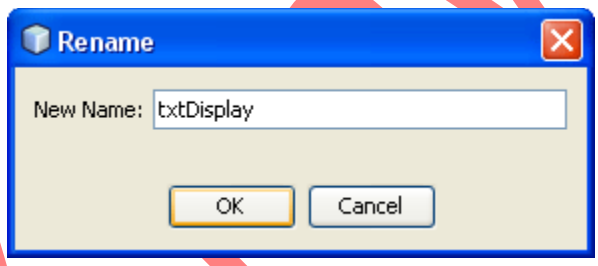
With the text field selected, have a look at the **Inspector** area in the bottom left: (If you can't see an Inspector area, click **Window > Navigating > Inspector** from the NetBeans menu bar.)



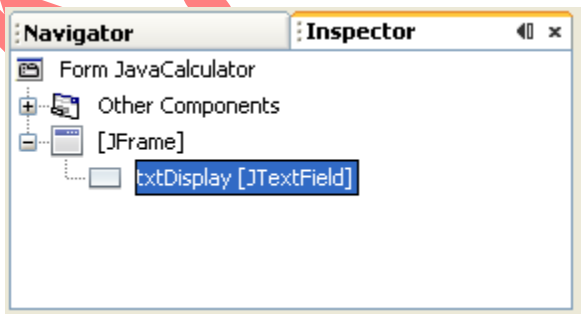
As you can see, **jTextField1** is selected in the Inspector. This area shows you what objects you have on your forms. You can also rename an object from here. To do so, right click on jTextField1. From the menu that appears, select **Change Variable Name**.



When you click on Change Variable Name, a dialogue box appears. Type a new name for the Text Field. Call it **txtDisplay**:



Click OK. When you do, NetBeans will rename your text field:



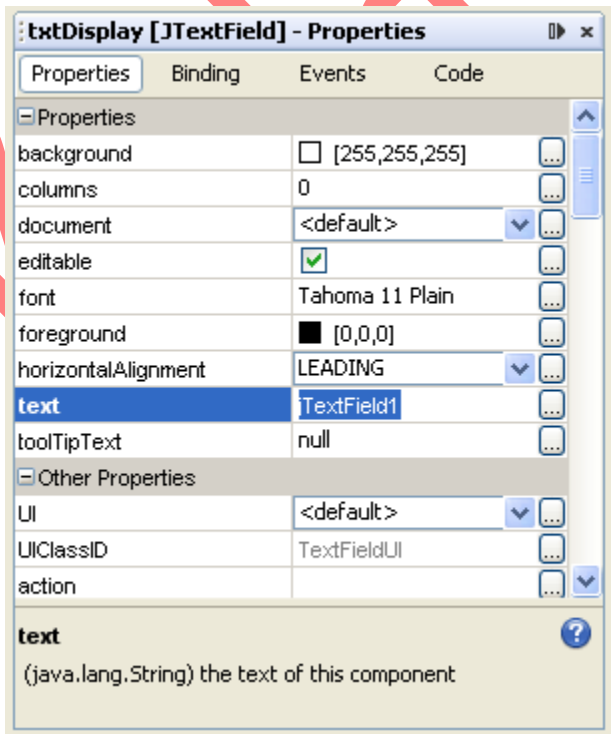
Now have a look at your code again by clicking the **Source** button in the main window. When your code appears, scroll down to the bottom. You'll see that a new private field variable has been added:

```
    /**...*/
    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new JavaCalculator().setVisible(true);
            }
        });
    }

    // Variables declaration - do not modify
    private javax.swing.JTextField txtDisplay;
    // End of variables declaration
```

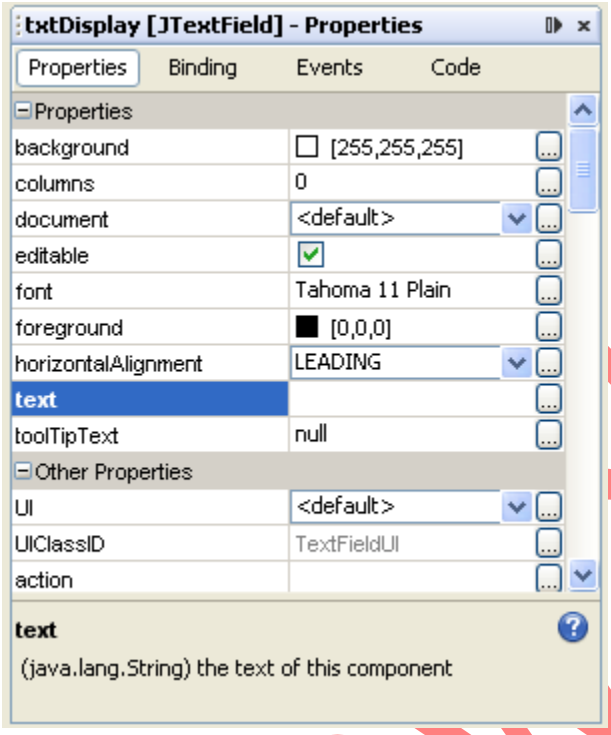
So a **JTextField** variable has been set up with the name **txtDisplay**. The "javax.swing" part is a reference to a set of packages that are used for GUI development. Swing makes it easier for you to create forms and the controls on forms.

Click on the **Design** button at the top to return to your form. The text field has some default text in, at the moment. You can add your own text by changing the text property of the text field.

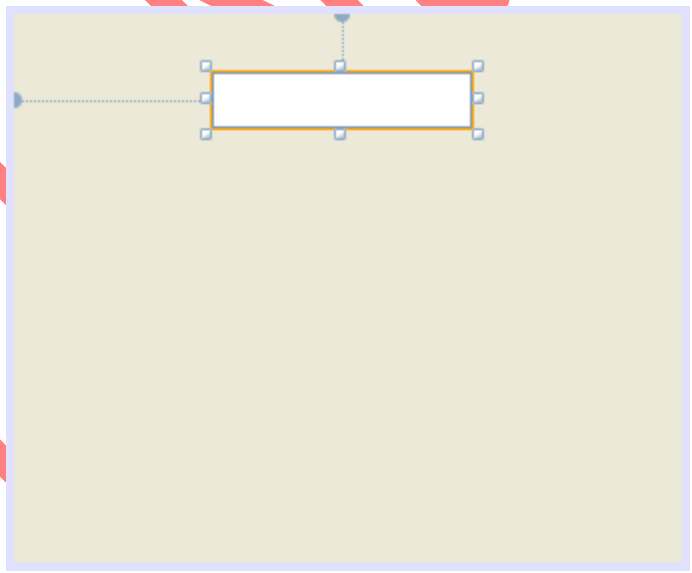
Make sure your text field is selected on the form. Now locate text in the properties window:



Delete the default text of **jTextField1** and leave it blank:



Then press the enter key on your keyboard. Have a look at your text field object on the form. It may well have changed size. Use the sizing handles to resize it. You'll see that it now has no text in it at all:

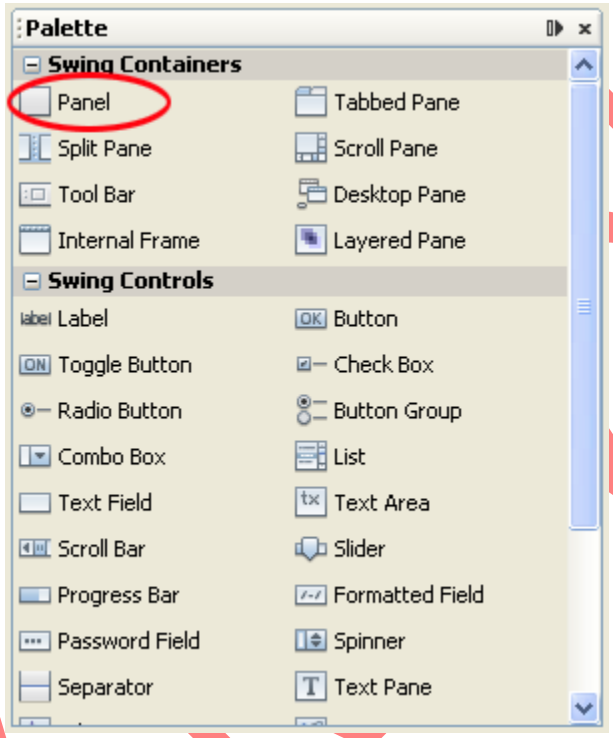


The text field on our calculator will obviously be used for the output of the calculation. But it won't work without buttons.

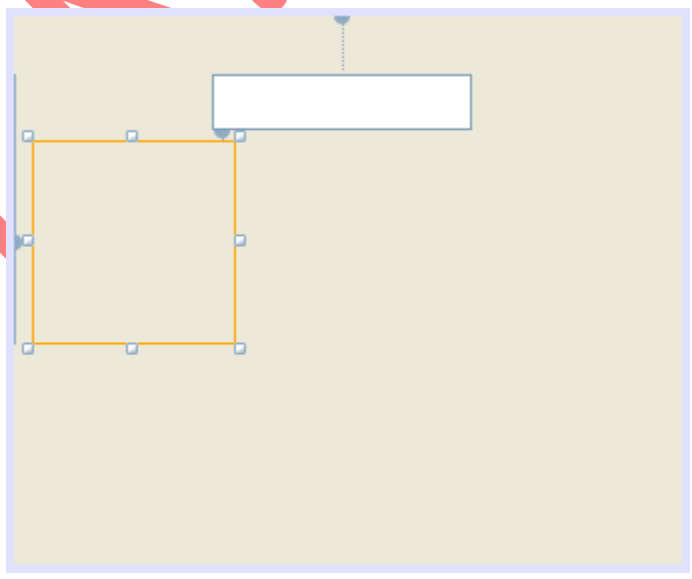
Add a Button to a Java Form

You add a button to a form in the same way you do for text fields - drag and drop. However, as we're going to be adding lots of buttons, it's a good idea to add the buttons to a control called a Panel. If you need to move the buttons, you can then just move the Panel instead. All the buttons will then move with the Panel.

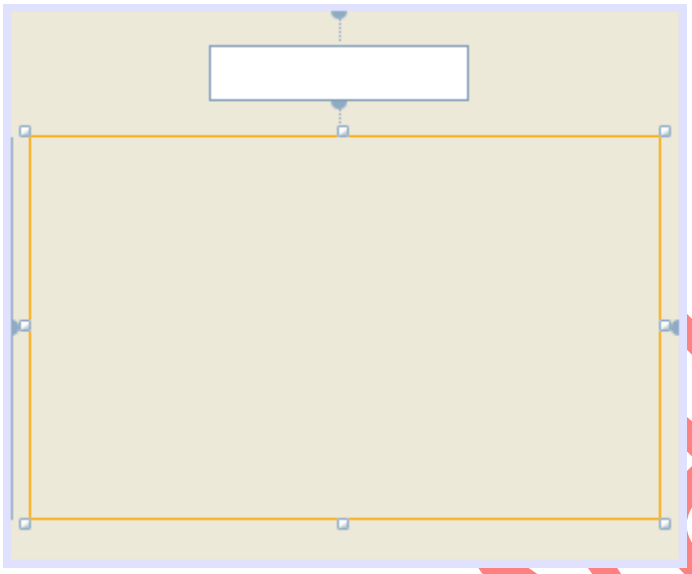
So locate the Panel control in the Palette:



Drag one on to your form. You can't see a Panel, as they'll have the same colour as the form. But you can select it:

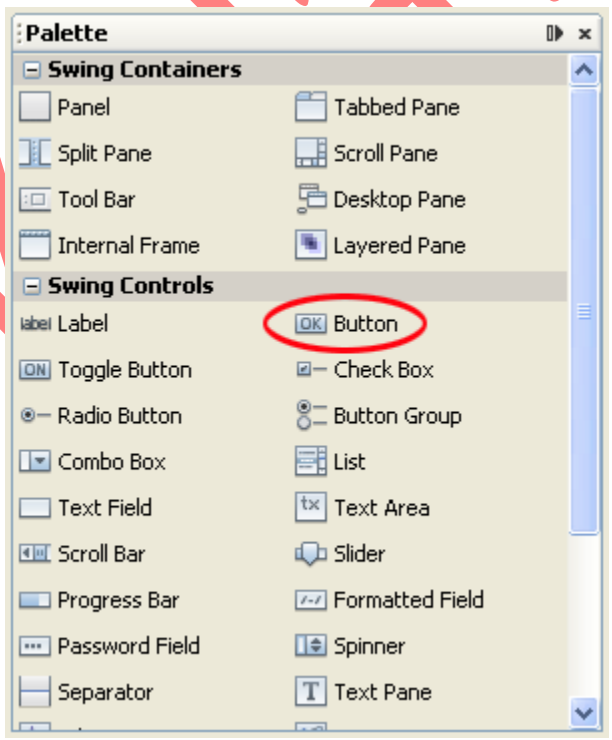


Drag the sizing handles so that the Panel fills most of the form:

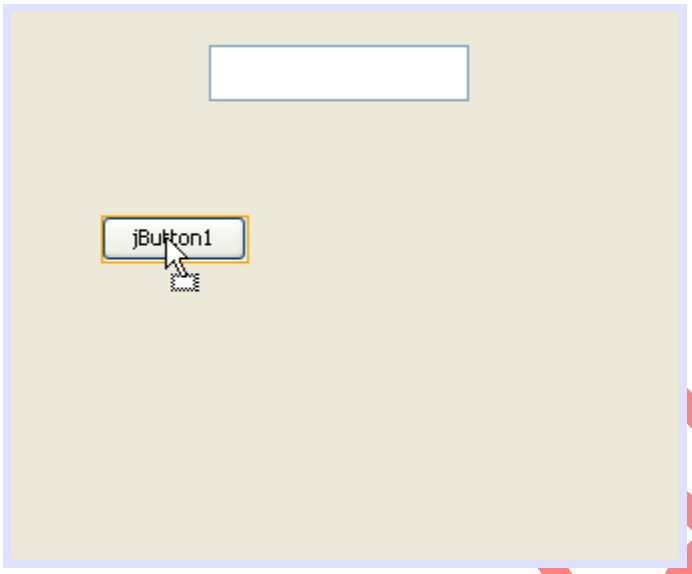


We can now add a button.

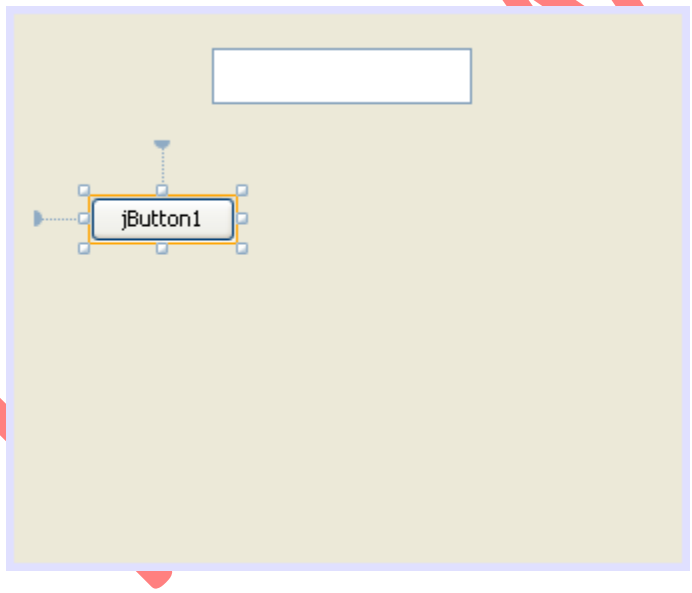
Locate the Button control in the Palette:



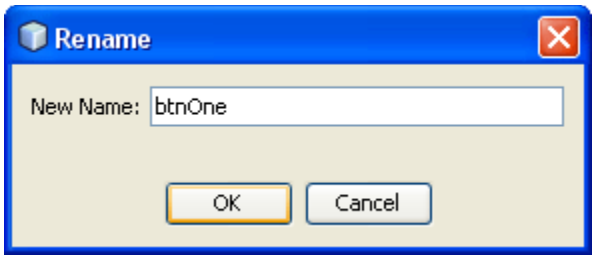
Drag one on to your Panel control:



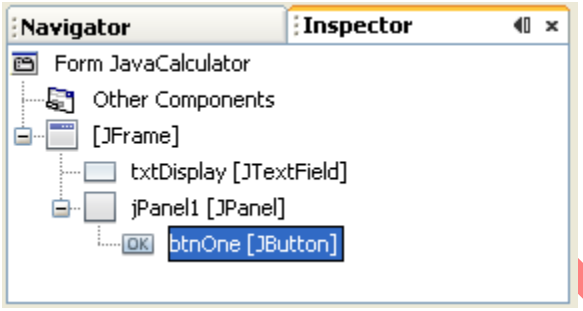
You should then see a button with sizing handles and position lines:



The default name of the button is **jButton1**. Change this name just like you did for the text field: right click in the Inspector area, and select **Change variable name**. Change the name of the button to **btnOne**:



The name of the button will have changed in the Inspector:



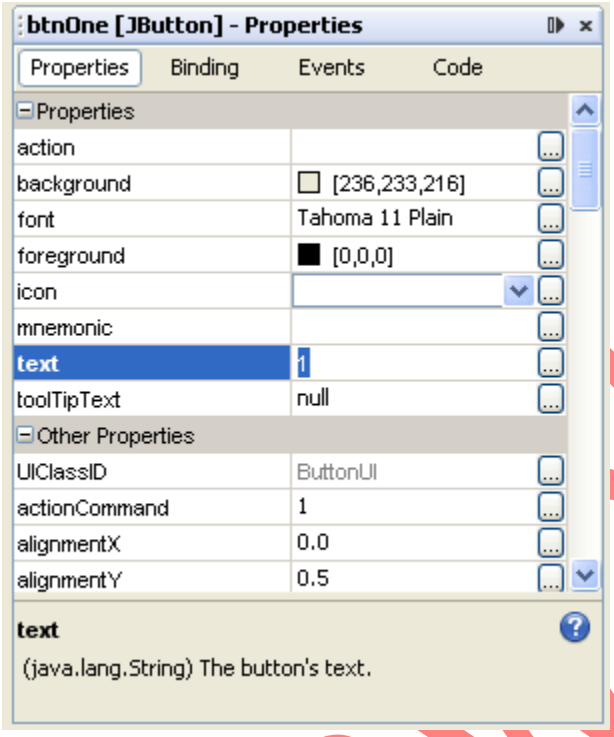
A new line of code has also been added:

```
    // Variables declaration - do not modify
    private javax.swing.JButton btnOne;
    private javax.swing.JPanel jPanel1;
    private javax.swing.JTextField txtDisplay;
    // End of variables declaration
```

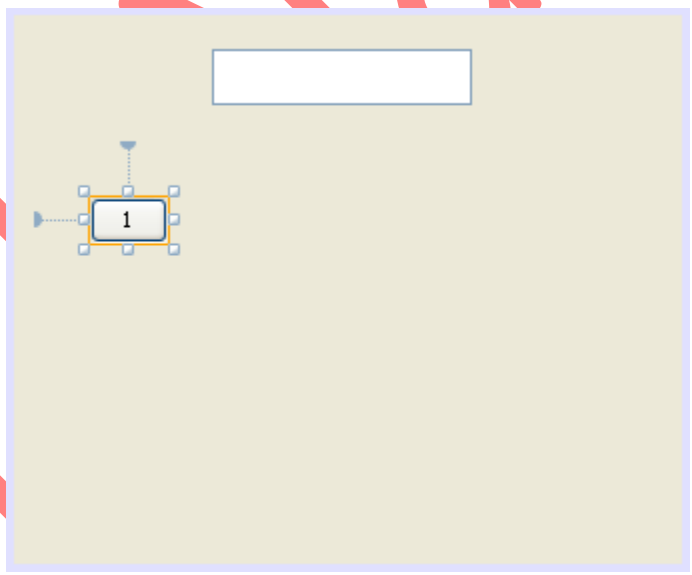
The new variable name is **btnOne**, and it is a **JButton** object, which is a Swing control. Note, too, that a variable has been set up for the panel (we've left this on the default name).

How to Change the Properties of a Button on a Java Form

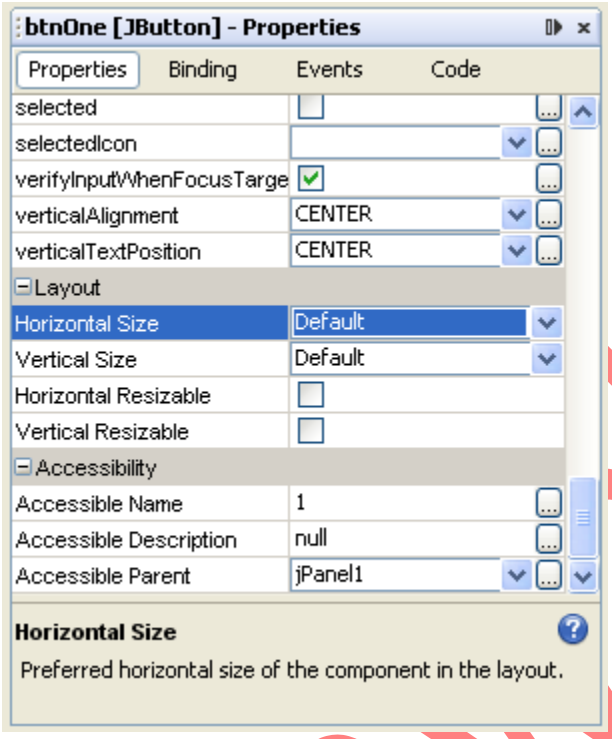
Go back to Design view and make sure your button is selected. We can change the text on the button. What we want is a number, one number for each button on the calculator. The **text** property is used for button text. So locate this property in the properties window. Delete the default and type the number 1 instead:



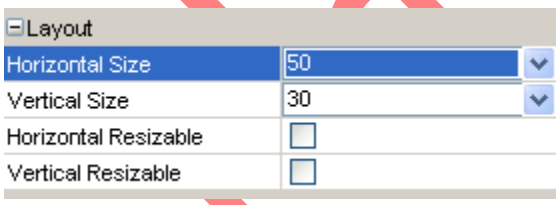
Press the enter key on your keyboard and you should see the text change on your button. It will also be resized:



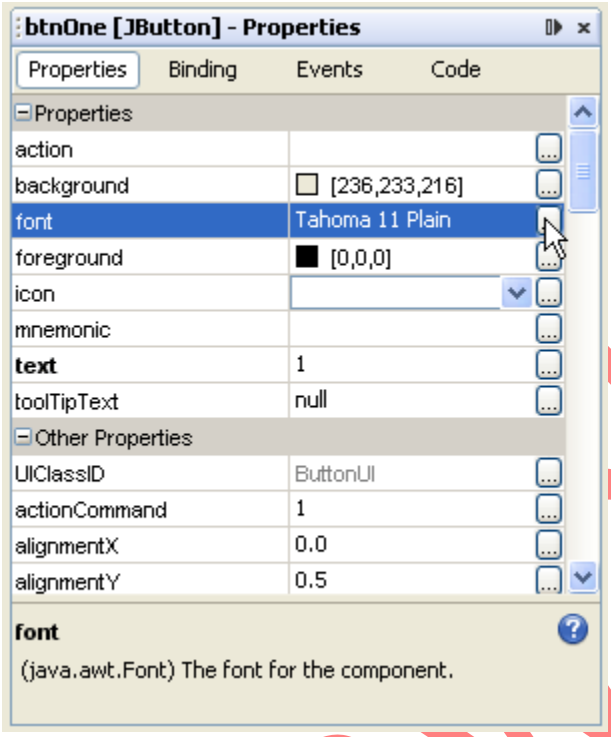
You can change the size of the button in the properties window. Locate **Horizontal Size** and **Vertical Size**, under the **Layout** heading:



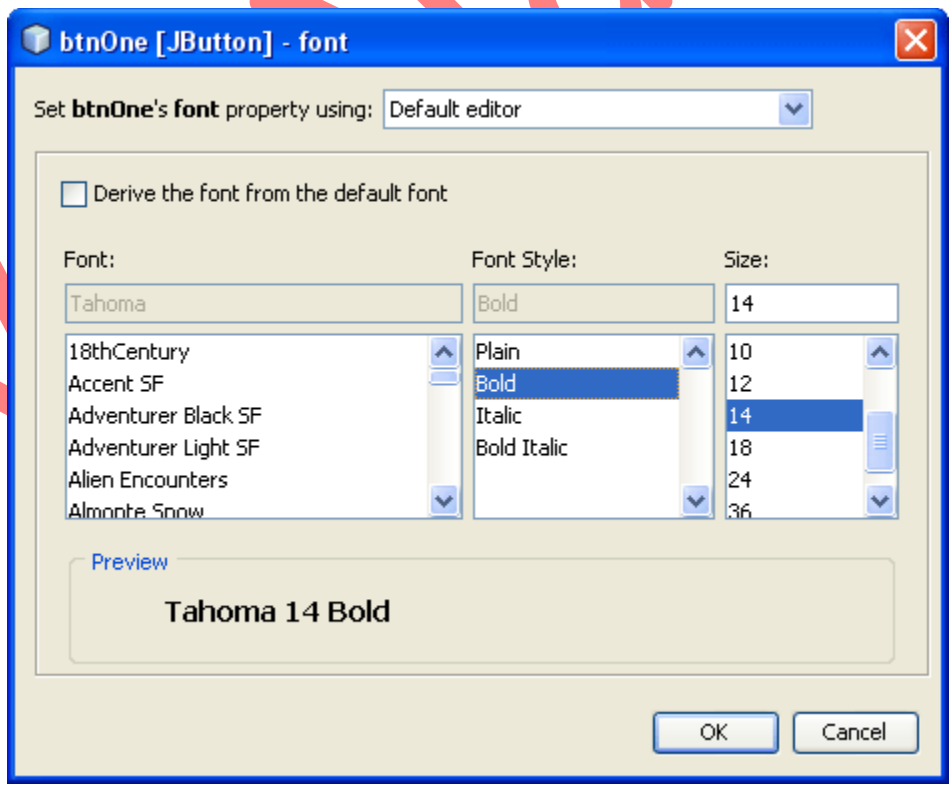
Change these values from the default to 50, 30:



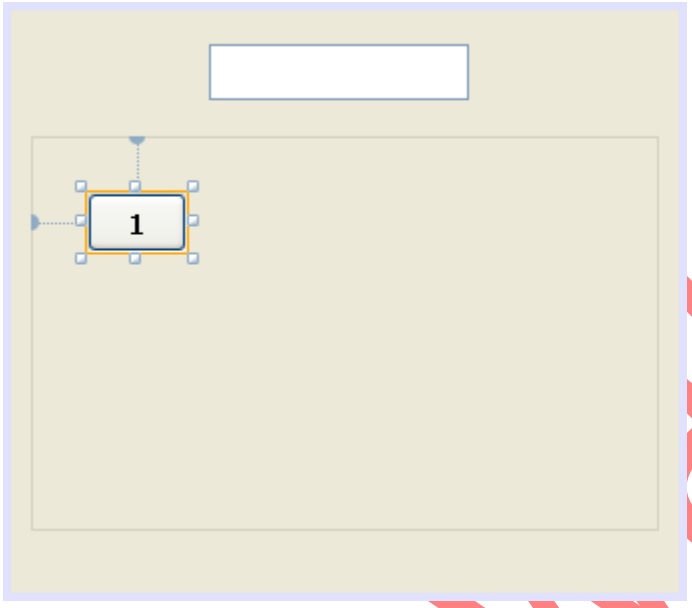
You can also change the font and font size in the properties window. Make sure your button is selected. Now locate the font property. Click the small button to the right of the font row:



When you click the font button, you should see a dialogue box appear. Change the font size to 14 points, and make it bold:

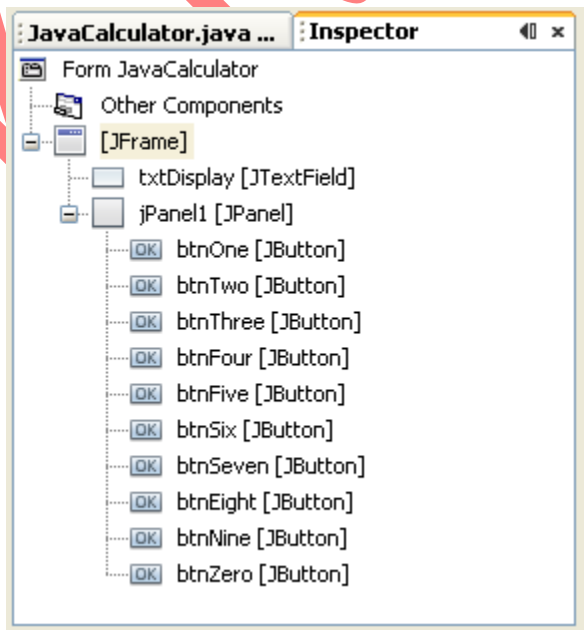


Your form should now look like this (to see the outline of the panel, just hover your mouse over it):



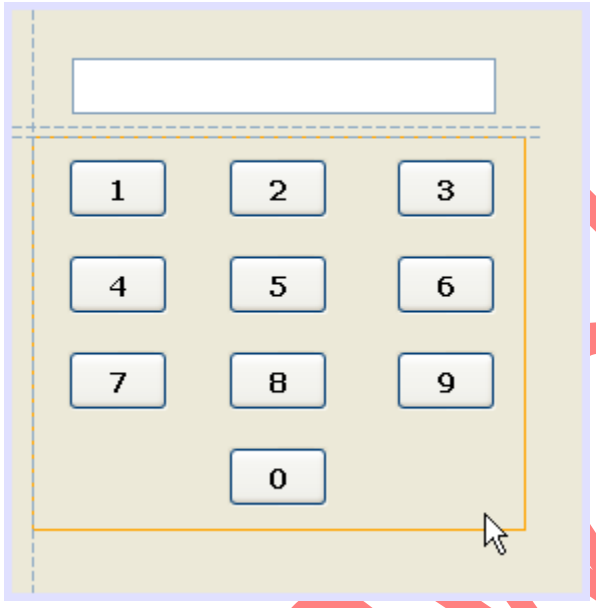
You now need to add nine more buttons in the same way, for the numbers 2 to 9, and a 0. Change each variable name to btnTwo, btnThree, btnFour, etc. Delete the default text, and enter a number instead. Then change the size of each button to 50, 30. Finally, change the font property of each button.

When you're finished, your Inspector area should look like this:

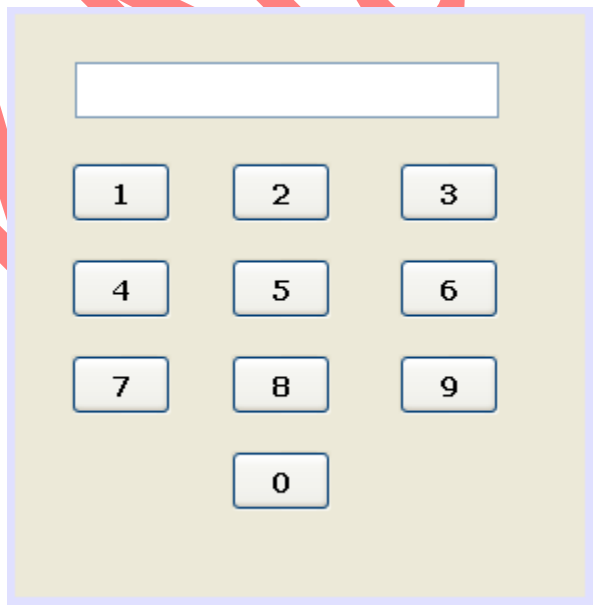


(Note that the 0 button has the name btnZero.)

If your buttons are in the wrong place, click a button to select it. Hold down your left mouse button. Keep it held down and drag to a new location in the panel. To move all the buttons at once, select the panel and drag it to a new location:



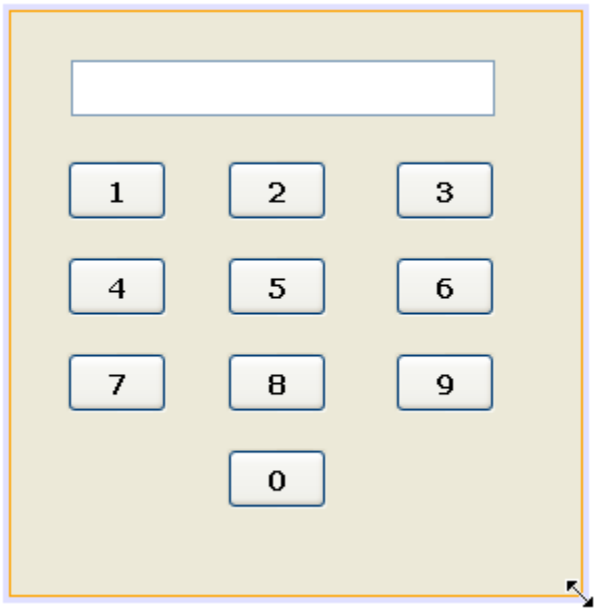
When you're happy with your form, it should look something like this (We've resized the text field):



Only three more buttons to add: a plus button, an equals button, and a clear button. We'll add these to a panel of their own, and move them to the right of the calculator.

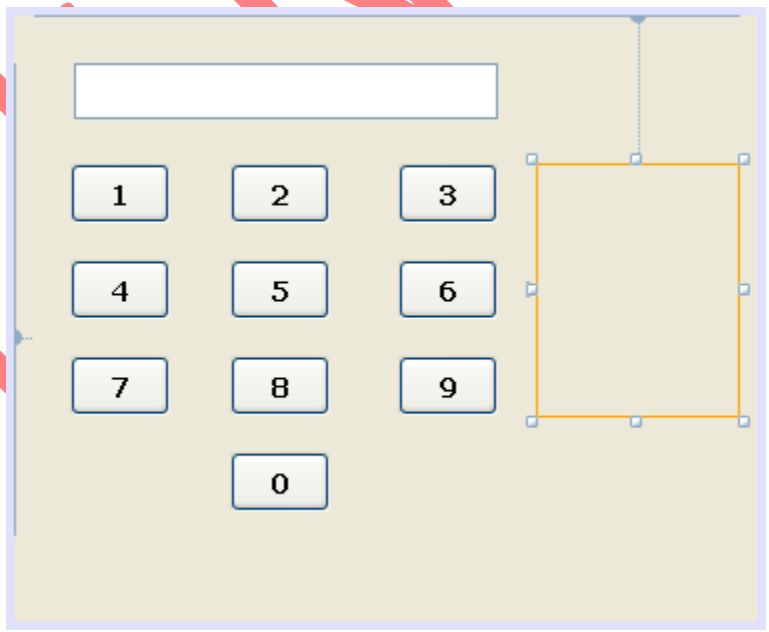
As you can see, our calculator doesn't have any room on the right. But you can easily resize a form.

Click on just the form, and not the panel or the text field. If you do it right, you should see an orange rectangle surrounding the form. Now move your mouse to the edges of the form. The mouse pointer should change shape, as in the image below (bottom right):



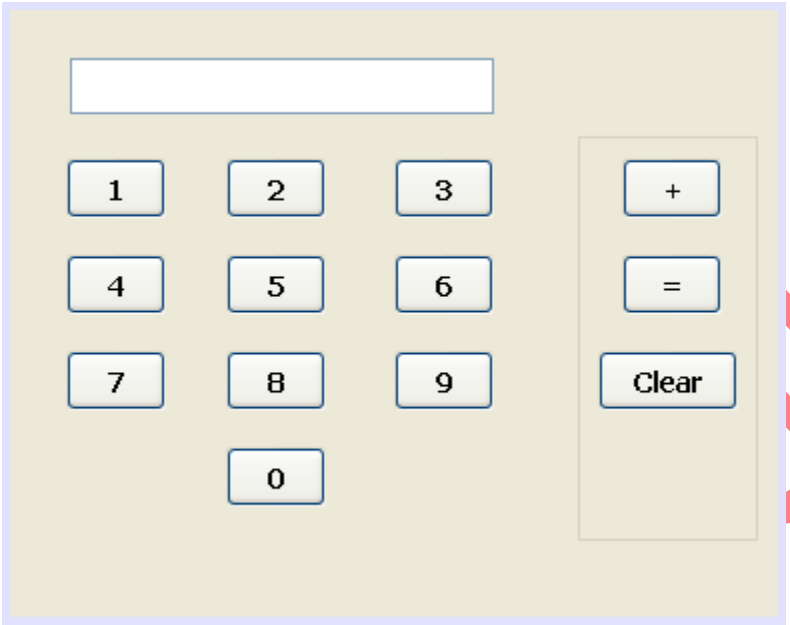
Hold down your left mouse button when the pointer changes shape. Now drag to a new size.

Add a panel in the space you've just created:

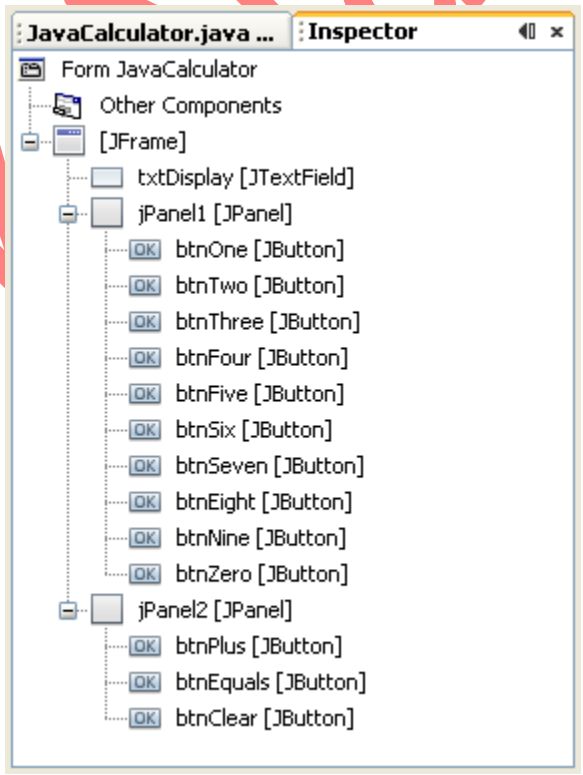


Add three buttons to the new panel. Change the variables names to: **btnPlus**, **btnEquals**, and **btnClear**. For the text for the buttons, type a + symbol for the Plus button, a = symbol for equals

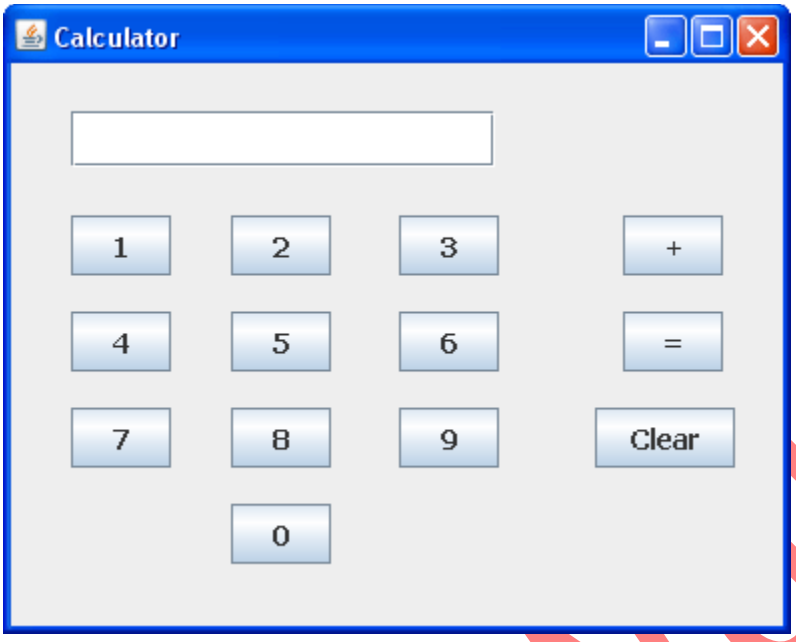
button, and the word "clear" for the clear button. Change the font to the same as the number buttons, 14 point bold. The size for the plus and equals button should be the same as the number buttons: 50, 30. For the Clear button, change it to 70, 30. Your form should then look like this:



The Inspector area of NetBeans should look like this:



You can run your form at this stage, just to see what it looks like:



Looking quite nice, hey! Nothing will happen when you click on the buttons, though. We'll add some code shortly. First, a word about Events.

Java Form Events

In programming terms, an event is when something special happens. For forms, this means things like buttons being clicked, the mouse moving, text being entered into text fields, the programming closing, and a whole lot more.

Events are objects in Java. They come from a series of classes stored in **java.util.EvenObject**. When a button is clicked, the button is said to be the source of the event. But the very act of clicking generates an event object. The event object then looks for an object that is listening out for, say, a mouse click, or a keystroke, or any other event that can happen on a form. A text field, for example, can be listening out for key strokes. Or a drop down box could be listening out for which item in the list was clicked.

Different objects (sources) fire different event. For a button, the event that is fired is the **ActionListener**. For a text field, it's the **KeyEvent**. Think of like this: the event object is responsible for passing messages back and forward between form objects that have had something happen to them, and objects that are waiting for things to happen.

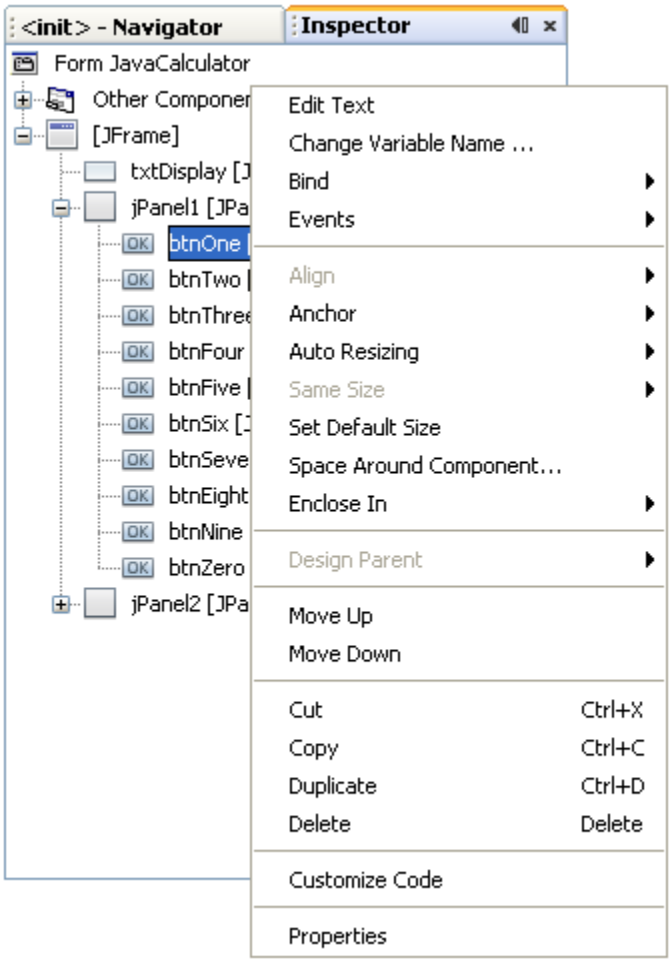
If all this sounds a bit complicated, then don't worry - it is complicated! But some programming examples might clear things up. First, a word about how our calculator will work.

If we want to add 3 + 2, we first need to click the 3 button. The number 3 will then appear in the text field. The plus button is clicked next, and this alerts the programme to the fact that we want to add things. It will also clear the text field ready for the next number. The next number is 2,

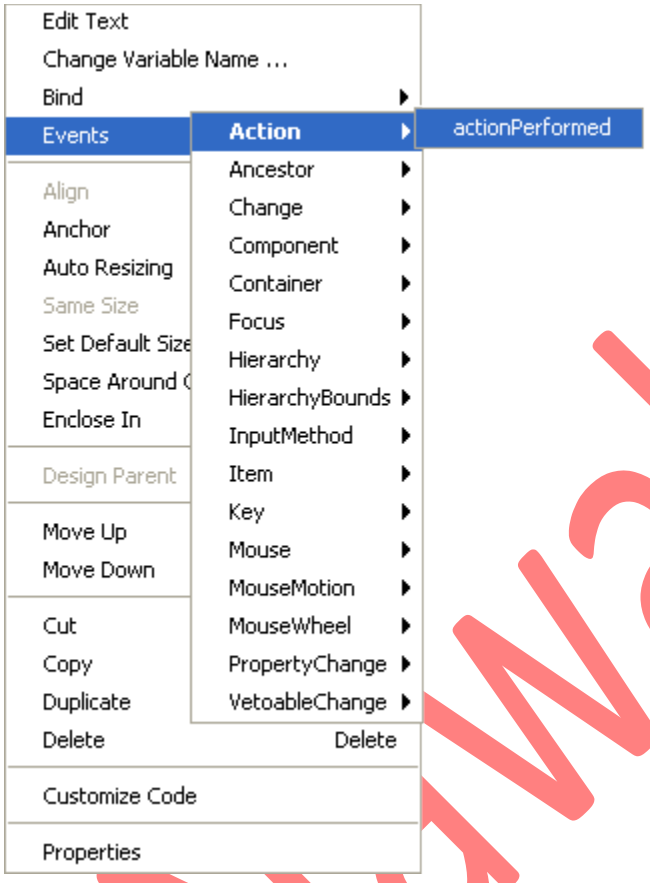
and we store this value along with the 3. The equals button is clicked to get the total, and this is where we take the stored numbers (3 and 2) and add them together. Finally, the answer is stored in the text field.

The first problem is how to get at the numbers on the buttons. We can do this by returning the text property of the button. Once we have the text, we can put it into the text box. But we need an ActionEvent for when the button is clicked.

In Design view in NetBeans, select your number 1 button. Now have a look at the Inspector window in the bottom left. Locate your btnOne button. Now right click to see the following menu appear:



Select **Event** from the menu. From the submenu, click on **Action**, and then **actionPerformed**:



When you click on **actionPerformed**, you will create a code stub for btnOne:

```
private void btnOneActionPerformed(java.awt.event.ActionEvent evt) {

// TODO add your handling code here:

}
```

The first line is a bit long. But it's just a method with an ActionEvent object between the round brackets. When the button is clicked, whatever code we write between the curly brackets will get executed.

Writing code for the numbers buttons on our Java Calculator

To get text from a form object, you can use the getText method of the object (if it has one). So to get the text from btnOne we could use this code:

```
String btnOneText = btnOne.getText( );
```

To get text from our text field, we could do this:

String textfieldText = txtDisplay.getText( );

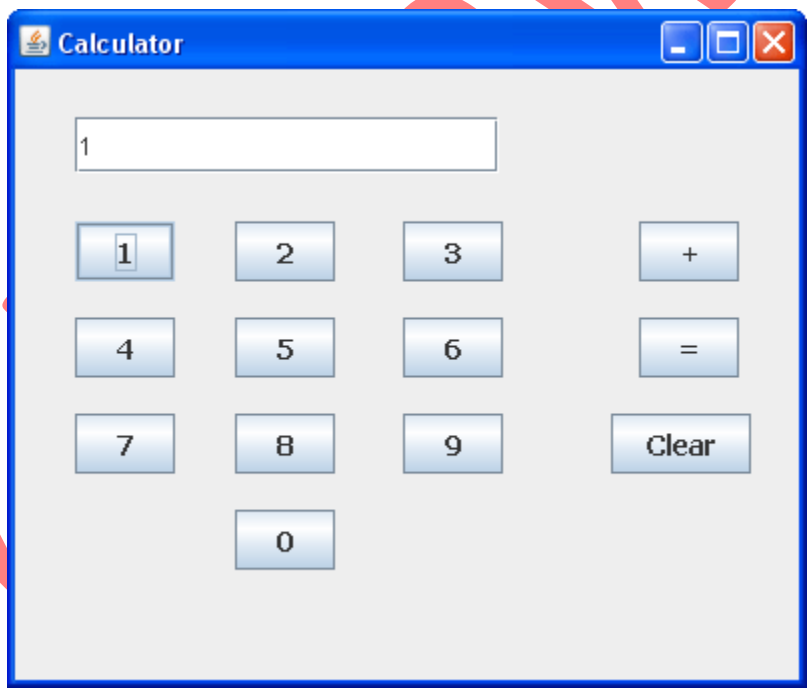
However, if wanted to put something into the text field, the method to use is setText:

txtDisplay.setText(btnOneText);

Try it out. Add this code to your code stub:

```
private void btnOneActionPerformed(java.awt.event.ActionEvent evt) {

String btnOneText = btnOne.getText( );
txtDisplay.setText(btnOneText);

}
```

Run your programme and test it out. Click your 1 button and you should find that the number 1 appears in the text field:



There is a problem, however. What if you want to enter the number 11, or the number 111? When you click the button repeatedly, nothing else happens. It's always a 1, no matter how many times you click.

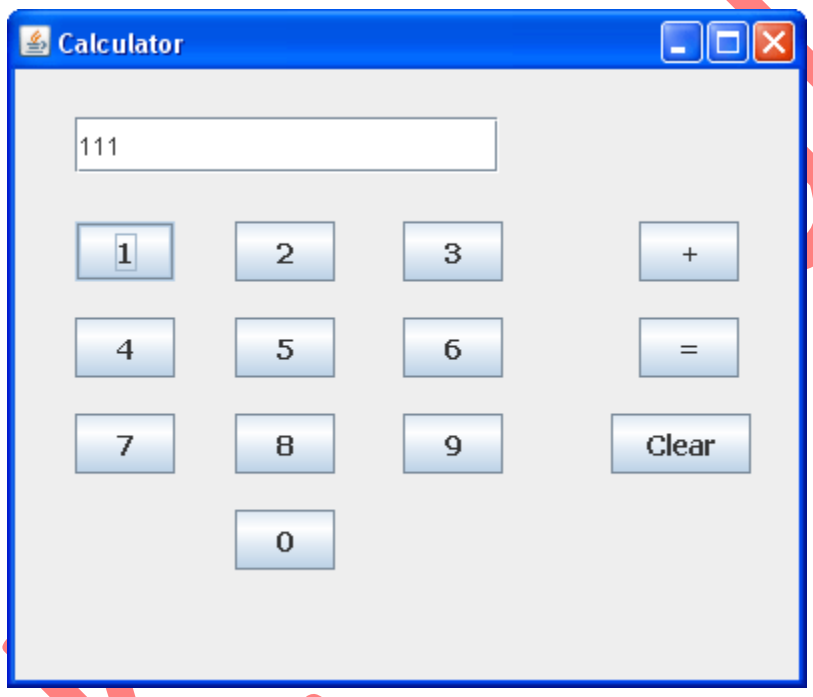
The reason is that our code doesn't keep whatever was in the text field previously. It just puts the number 1 there. If we wanted the number 11, we would have to keep the first 1. To do that, you

can simply get the text from the text field and combine it with the button text. Amend the first line of your code to this:

String btnOneText = txtDisplay.getText() + btnOne.getText();

Now we're saying get the text from the text field and combine it with the button text. Store the result in the variable called btnOneText.

Run your programme again, and click your number 1 button a few times. You should find that the text field will display a series of 1's:



We can add the same code for all the number buttons on the calculator. Go back to design view. Instead of right-clicking in the Inspector area, another way to add an Action for a button is to simply double click it. So double click your number 2 button. A code stub will be created, just like last time.

Add the following code to it:

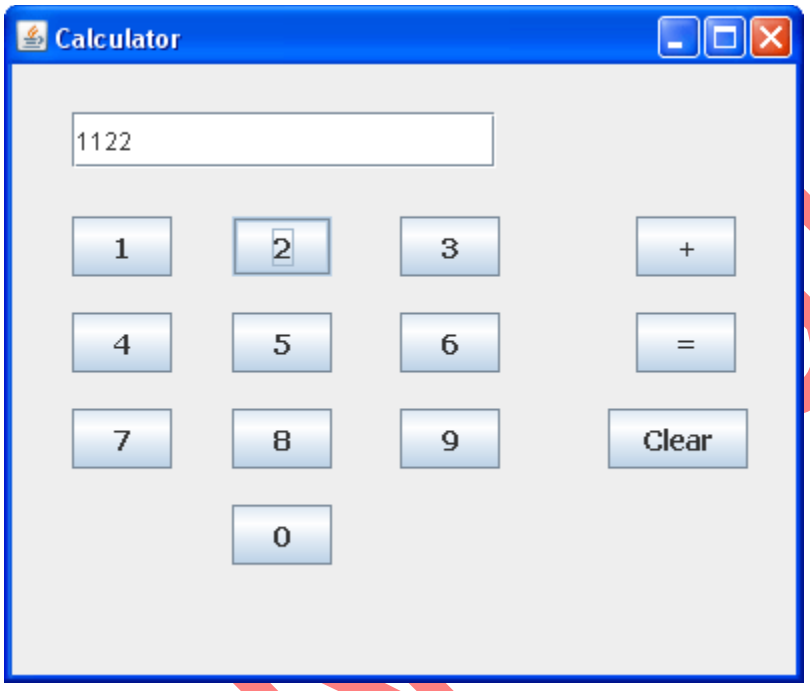
private void btnTwoActionPerformed(java.awt.event.ActionEvent evt) {

```
String btnTwoText = txtDisplay.getText() + btnTwo.getText();
txtDisplay.setText( btnTwoText );
```

}

The only difference with the code is the name of the String variable (now called **btnTwoText**), and the fact that we're getting the text from **btnTwo**. In between the round brackets of **setText**, we pass the name of the String variable.

Run your programme again. You should now be able to click the 1 and the 2 buttons and have the text appear in the text field:



Add the same code for all of your calculator buttons. So double click each button to get the code stub, and add the two lines of code for each button. You will need to change the name of the String variable. Here's the code for button 3:

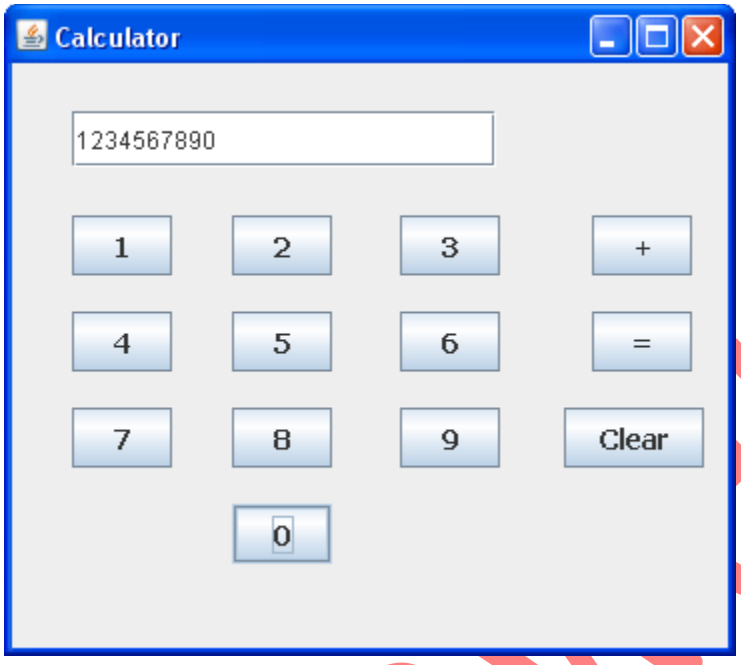
String btnThreeText = txtDisplay.getText() + btnThree.getText();
txtDisplay.setText(btnThreeText);

You can copy and paste the code you already have. Then just change the String variable name, the name of the button after the btn part, and the part between the round brackets of setText. If you see any underlines then you know you've done something wrong.

When you've finished, the code for all the number button should look like this:

```
private void btnOneActionPerformed(java.awt.event.ActionEvent evt) {

    String btnOneText = txtDisplay.getText() + btnOne.getText();
    txtDisplay.setText(btnOneText);
}

private void btnTwoActionPerformed(java.awt.event.ActionEvent evt) {

    String btnTwoText = txtDisplay.getText() + btnTwo.getText();
    txtDisplay.setText(btnTwoText);
}

private void btnThreeActionPerformed(java.awt.event.ActionEvent evt) {

    String btnThreeText = txtDisplay.getText() + btnThree.getText();
    txtDisplay.setText(btnThreeText);
}

private void btnFourActionPerformed(java.awt.event.ActionEvent evt) {
    String btnFourText = txtDisplay.getText() + btnFour.getText();
    txtDisplay.setText(btnFourText);
}

private void btnFiveActionPerformed(java.awt.event.ActionEvent evt) {
    String btnFiveText = txtDisplay.getText() + btnFive.getText();
    txtDisplay.setText(btnFiveText);
}

private void btnSixActionPerformed(java.awt.event.ActionEvent evt) {
    String btnSixText = txtDisplay.getText() + btnSix.getText();
    txtDisplay.setText(btnSixText);
}

private void btnSevenActionPerformed(java.awt.event.ActionEvent evt) {
    String btnSevenText = txtDisplay.getText() + btnSeven.getText();
    txtDisplay.setText(btnSevenText);
}

private void btnEightActionPerformed(java.awt.event.ActionEvent evt) {
    String btnEightText = txtDisplay.getText() + btnEight.getText();
    txtDisplay.setText(btnEightText);
}

private void btnNineActionPerformed(java.awt.event.ActionEvent evt) {
    String btnNineText = txtDisplay.getText() + btnNine.getText();
    txtDisplay.setText(btnNineText);
}

private void btnZeroActionPerformed(java.awt.event.ActionEvent evt) {
    String btnZeroText = txtDisplay.getText() + btnZero.getText();
    txtDisplay.setText(btnZeroText);
}
```

Run your calculator and test it out. You should be able to enter all the numbers from 0 to 9:



In the next part, you'll write the Java code for the Plus button on your calculator.

Coding the Plus Button of our Java Calculator

Now that the number buttons are working on your Java calculator, the next thing to do is to handle the plus button. The only thing our plus button needs to do is to store whatever number is currently in the text field. It's just a record of the first number to be added. Later, we'll also record the fact that it was the plus button that was clicked, rather than the minus, or the divide, or the multiply button.

To store the value, we need to set up a field variable, that's a variable outside of any button code. This is so that all the buttons can see what has been stored in it.

Add the following variable near the top of the coding window: (You can place it near the bottom, with the field variables NetBeans has set up for the form objects, but we'll keep our variables separate.)

private double total1 = 0.0;

Here's the code window:

```
public class JavaCalculator extends javax.swing.JFrame {

    private double total1 = 0.0;

    /** Creates new form JavaCalculator */
    public JavaCalculator() {...}

    /**...*/
```

So we're setting up a variable called **total1**. The type of variable is a **double**. Its default value is 0.0.

To store the value from the text field, we need to get the text. But we'll need to convert it from a String to a Double. You can do so with the parseDouble method of the Double object:

Double.parseDouble(txtDisplay.getText( ))

In between the round brackets of parseDouble we get the text from the txtDisplay text field.

However, when we store the value from the text field into the **total1** variable, we need to retain what is already in total1. We can also clear the text field, ready for the second number.

So return to Design view in NetBeans. Double click your Plus button to generate the code stub. Now add the following two lines to your plus button:

total1 = total1 + Double.parseDouble(txtDisplay.getText( )) ;
txtDisplay.setText("");

In between the round brackets of setText, we have a pair of double quotes, with no space between them. This is enough to clear the text field.

And that's it for the Plus button, for the moment. We'll come back to it later. All we're doing, though, is storing a number in the total1 variable, and keeping what is already there. Once the number is stored, we've cleared the text field. The user can now enter a second number to be added to the first.

Coding the Equals Button of our Java Calculator

After the user has chosen a second number, the equals button needs to be clicked. Clicking the equals button will produce the answer to the addition.

To store the output of the calculation, we can set up another field variable. Add the following line to the top of your code:

private double total2 = 0.0;

Your code window should look like this:

```
public class JavaCalculator extends javax.swing.JFrame {

    private double total1 = 0.0;
    private double total2 = 0.0;

    public JavaCalculator() {
        initComponents();
    }
```

To get the answer to the calculation, we take whatever is currently stored in total1 and add it to whatever is currently in the text field. Again, though, we need to parse the string from the text field and turn it into a double.

Go back to Design view and double click your equals button. In the code stub that is created, add the following line:

total2 = total1 + Double.parseDouble(txtDisplay.getText( ));

This line gets the text from the text field and converts the string into a double. The result is then added to total1. The answer is then stored in the total2 variable.

The next thing we need to do is to display the answer to the calculation back into the text field. However, we now have to convert the double back into a string, as text fields hold text and not numbers. If you try to store a double value directly into a text field you'll get errors.

To convert a double into text you can use the toString method of the Double object. Add the following line just below the first one:

txtDisplay.setText(Double.toString(total2));

The conversion is done between the round brackets of setText. But you could set up a new variable, if you wanted:

String s1 = Double.toString(total2);
txtDisplay.setText(s1);

But the result is the same: convert a double to a string.

The final line for the equals button can clear the total1 variable. Once the total1 variable is cleared, a new calculation can be entered. Here's the line to add:

total1 = 0;

The three lines for your equals button should now be these:

```
total2 = total1 + Double.parseDouble( txtDisplay.getText( ) ) ;
txtDisplay.setText( Double.toString(total2) );
total1 = 0;
```

Coding the Clear Button of our Java Calculator

The only thing left to code for now is the Clear button. For this, we need to clear the **total2** variable and set the text in the text field to a blank string.

In Design view, double click your Clear button to create the code stub. Now add these two lines to your Clear button:

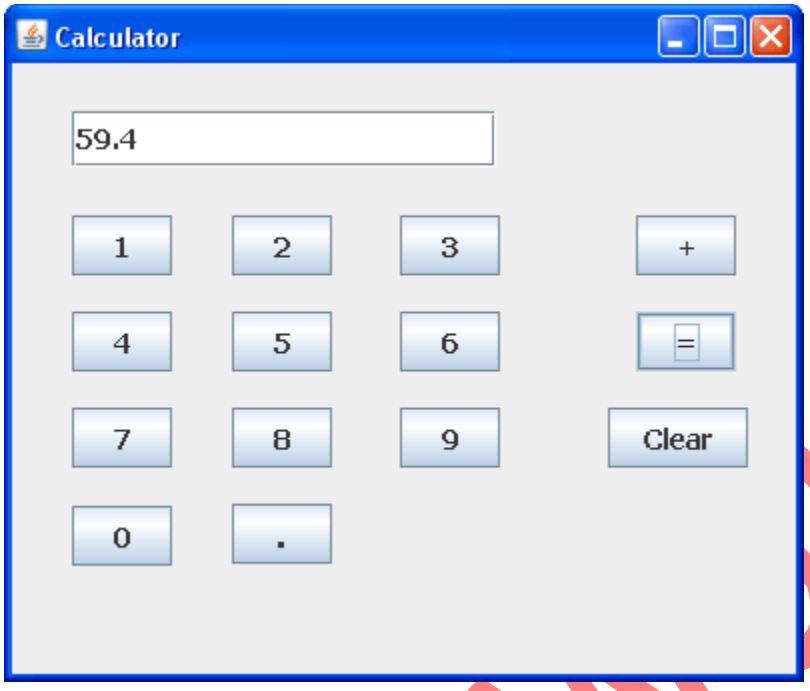
```
total2 = 0;
txtDisplay.setText("");
```

Once you've added the lines, you can try your calculator out. Run the programme and try adding numbers. You should find that your calculator adds up fine.

Exercise
There is one thing missing, however - a point symbol. At the moment, your calculator can't handle sums like 23.6 + 35.8. So add a point button to your form. Write the code for it. (Hint - it's practically the same code as for your number buttons.)

Exercise
Set the font property for your text field so that it's 14 point bold

When you've completed the above exercises, you calculator should look something like ours:



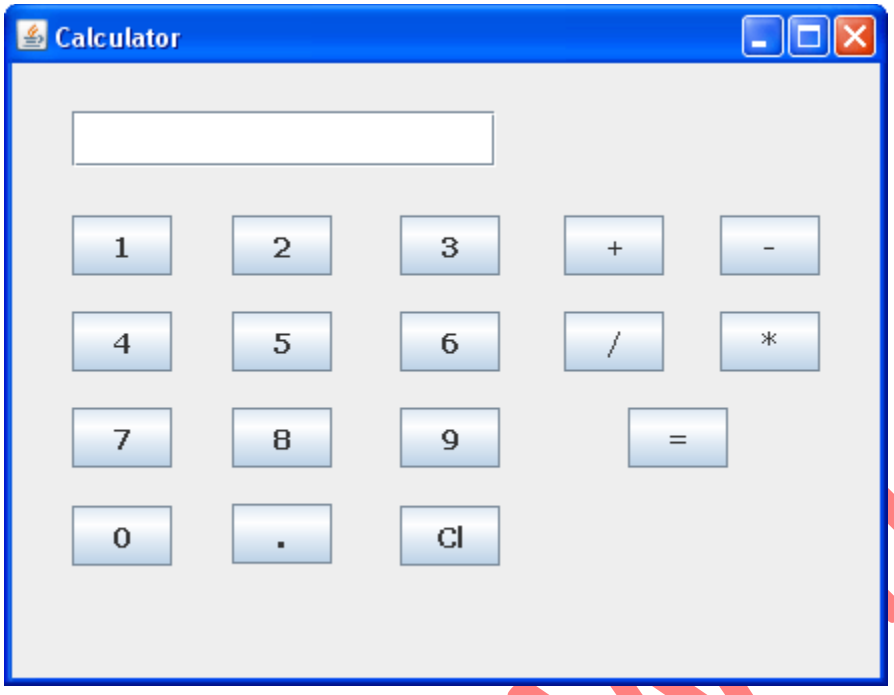
Congratulations - you have now written a Java calculator programme! OK, it can only add up, but it's a start.

What we'll do now is to make it subtract, divide and multiply.

Subtract, Divide and Multiply Buttons for our Java Calcualtor

Now that the plus button is working on our Java Calculator, we can add buttons to subtract, divide and multiply. Just like the addition button, however, these buttons won't do any calculations: the equals button will still do all the work. The only thing the operator buttons will do is to record which button was clicked: add, subtract, divide, or multiply.

The first thing to do is to place some more buttons on your form. In the image below, we've moved the Clear button, and put all the operator buttons in the panel to the right. Feel free to come up with your own design, though:



Once you've added the new buttons, rename the default variables to btnSubtract, btnDivide, and btnMultiply. (Another way to rename the variable is to simply right-click the button. You'll then see a menu appear. Select "Change Variable Name".)

The technique we'll use to get which operator button was clicked is to store the button text in a field variable. We can then use a switch statement to examine which character is in the field variable. If it's the + symbol we can add; if it's the ? symbol we'll subtract; if it's the / symbol we'll divide; and if it's the * symbol we'll multiply.

So click the Source button to get back to your code. Add the following field variable to the top, just below your other two:

private char math_operator;

The top of your code should then look like this:

```
public class JavaCalculator extends javax.swing.JFrame {

    private double total1 = 0.0;
    private double total2 = 0.0;
    private char math_operator;

    public JavaCalculator() {
        initComponents();
    }
```

We can set up a method to get at the character on the button that was clicked. Add the following method to your code:

```
private void getOperator(String btnText) {
  math_operator = btnText.charAt(0);
  total1 = total1 + Double.parseDouble(txtDisplay.getText());
  txtDisplay.setText("");
}
```

You can add the above method anywhere in your code, as long as it's between the curly brackets of the Class, and not between the curly brackets of any other method.

We've called the method getOperator. It's a void method, so won't return any value - it will just get on with executing the code. In between the round brackets of the method header, we have a String variable called btnText. This is obviously the text from the button that was clicked.

The text from the button is a string. However, switch statements in Java can't handle strings, so we need to convert the string to a character. We do so with this line:

math_operator = btnText.charAt(0);

The charAt method of strings will get a character from a string. The character you want goes between the round brackets of charAt. The math symbol from our buttons is always at character 0 in the string. This is then stored in the char field variable we've just set up at the top of the code.

Notice the other two lines in the code. They are exactly the same as the lines from the plus button, doing exactly the same thing - storing the first number in the variable called total1. Each operator button needs to do this, so it makes sense to have these two lines in our method, rather than in the operator button code.

So locate your btnPlus code and delete the following two lines from it:

total1 = total1 + Double.parseDouble(txtDisplay.getText( ));
txtDisplay.setText("");

Replace them with these two lines:

String button_text = btnPlus.getText();
getOperator(button_text);

The first new line gets the text from the plus button and stores it in a string variable. This is then handed over to our method getOperator.

The same two lines can be added to the other operator button, only changing the name of the button.

Go back to design view and double click your subtract button. For the code stub, add the following:

```
String button_text = btnMinus.getText();
getOperator(button_text);
```

(Although we've used the same name for the String variable Java won't get confused, as each button_text is local to its particular button code.)

Double click your divide button and add this:

```
String button_text = btnDivide.getText();
getOperator(button_text);
```

And here's the code for the multiply button:

```
String button_text = btnMultiply.getText();
getOperator(button_text);
```

Now that we have code for all four operator button, we can adapt the equals button.

For the equals button, we can set up a switch statement to examine what is in the math_operator variable.

```
switch ( math_operator ) {
case '+':
break;
case '-':
break;
case '/':
break;
case '*':
break;
}
```

The switch statement has a case for each of the math operators: +, -, /, and *. We haven't added any code, yet. But have a look at the code you already have for you equals button:

```
total2 = total1 + Double.parseDouble( txtDisplay.getText( ) ) ;
txtDisplay.setText( Double.toString(total2) );
total1 = 0;
```

The last two lines are OK, and don't need to be changed. The first line, however, can be used in the switch statement. This line, remember, is the one that adds up.
It can be moved up and used as the code for the + case:

case '+':
total2 = total1 + Double.parseDouble(txtDisplay.getText( ));
break;

If the minus button was clicked, then we can simply change the plus to a minus in the line above:

case '-':
total2 = total1 - Double.parseDouble(txtDisplay.getText( ));
break;

The case for the divide is then this:

case '/':
total2 = total1 / Double.parseDouble(txtDisplay.getText( ));
break;

And the case for the multiply character is this:

case '*':
total2 = total1 * Double.parseDouble(txtDisplay.getText( ));
break;

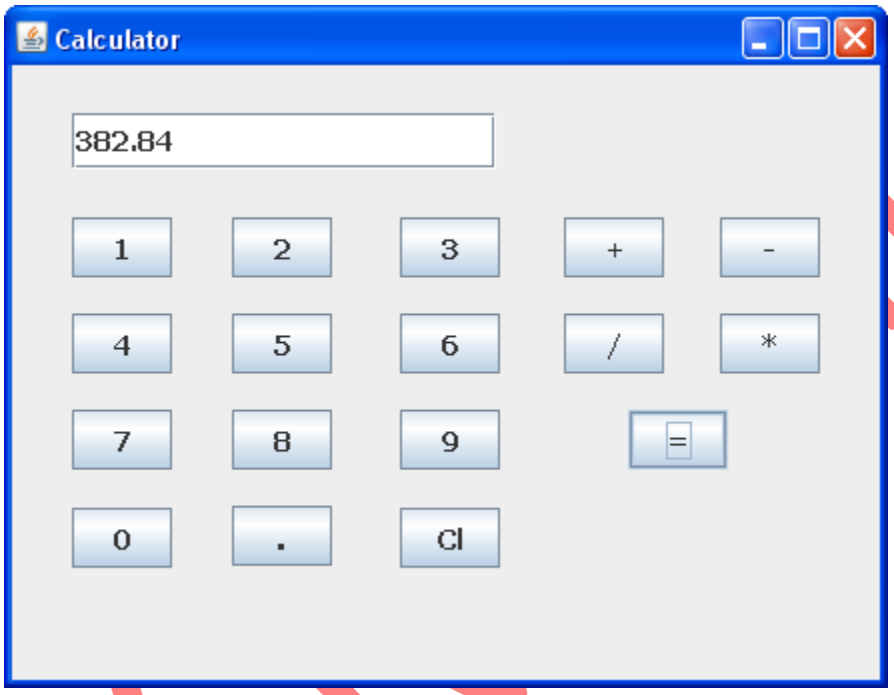
Here's the entire code for your equals button:

```
switch ( math_operator ) {
  case '+':
    total2 = total1 + Double.parseDouble(txtDisplay.getText());
    break;
  case '-':
    total2 = total1 - Double.parseDouble(txtDisplay.getText());
    break;
  case '/':
    total2 = total1 / Double.parseDouble(txtDisplay.getText());
    break;
  case '*':
    total2 = total1 * Double.parseDouble(txtDisplay.getText());
    break;
}

txtDisplay.setText(Double.toString(total2));
total1 = 0;
```

Once you've added the new code to your equals button, run your calculator and try it out. Try the following, to see if it works:

58. 6 + 37.5 (answer should be 96.1)
78 - 25.5 (answer should be 52.5)
68 / 8 (answer should be 8.5)
56.3 * 6.8 (answer should be 382.84)

And that's it - you now have a simple working calculator that can add, subtract, divide, and multiply:



Now that you've had some practice with form objects, let's create a new programme that makes use of the more common controls you'll find on a form.

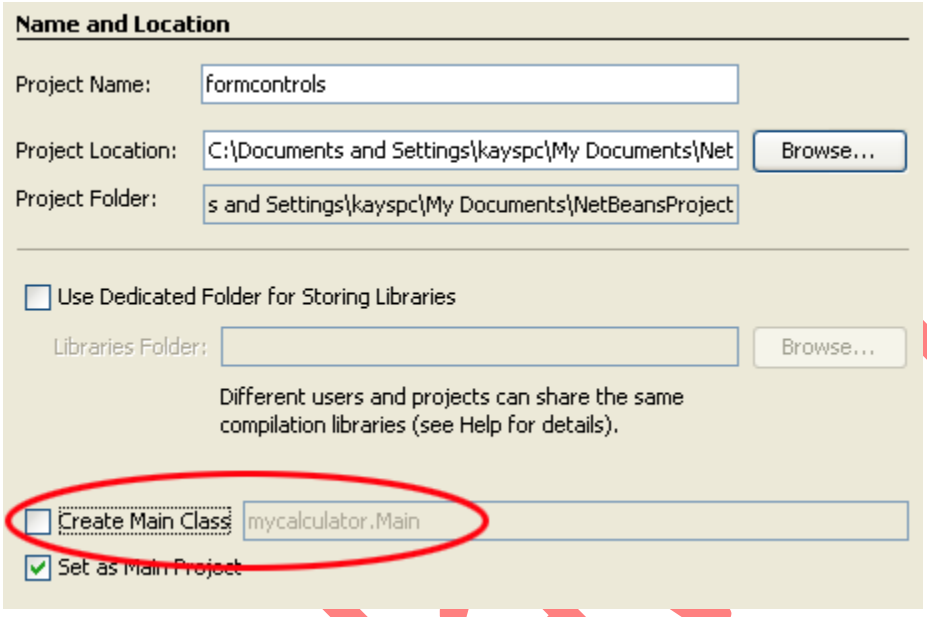
Java Combo Boxes

In this section, you'll see how to use some of the more common controls you can add to a Java form. You'll learn how to use the following:

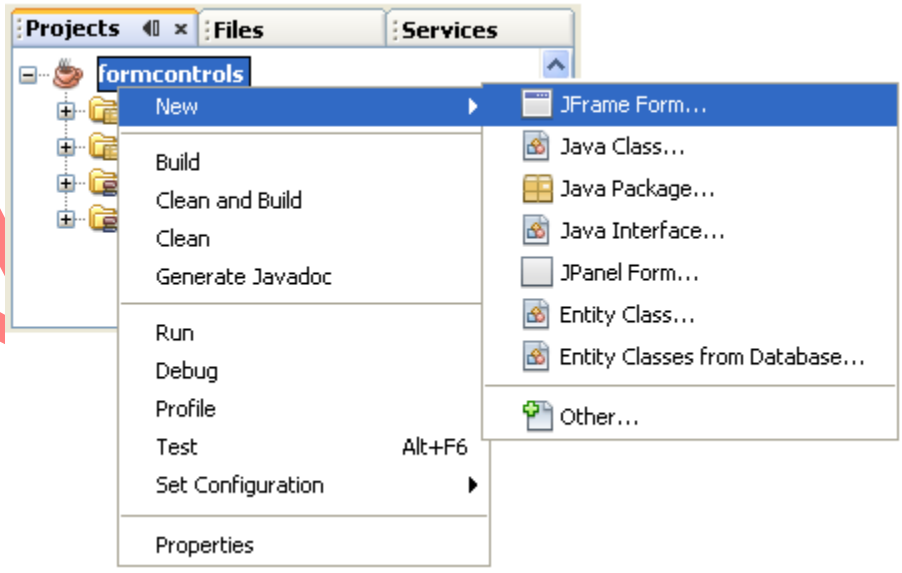
Combo Box
Check Box
Radio Buttons
Text Areas
List Box
Menus and Menu Items
Open File Dialogue boxes
Save File Dialogue boxes

We'll start with Combo Boxes.

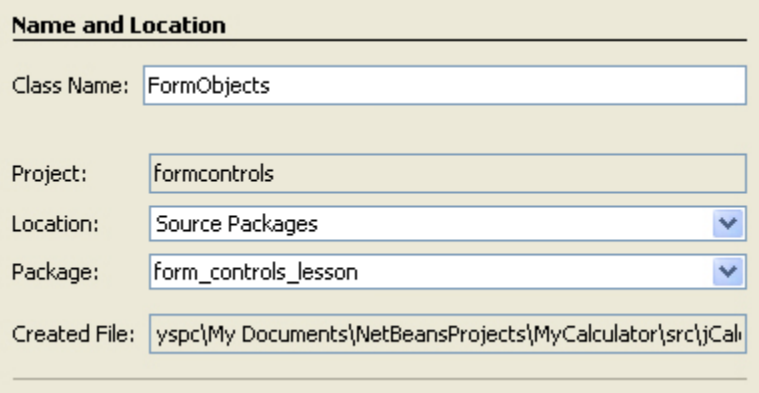
Create a new project for this (Java > Application). Call the project **formcontrols**, and uncheck the "Create main class" box:



Click the Finish button on the wizard to create the project. Now add a form by right-clicking the project name in Projects window and selecting **New > JFrame Form**:



When the dialogue box appears, enter **FormObjects** as the Class name, and **form_controls_lesson** as the package name:

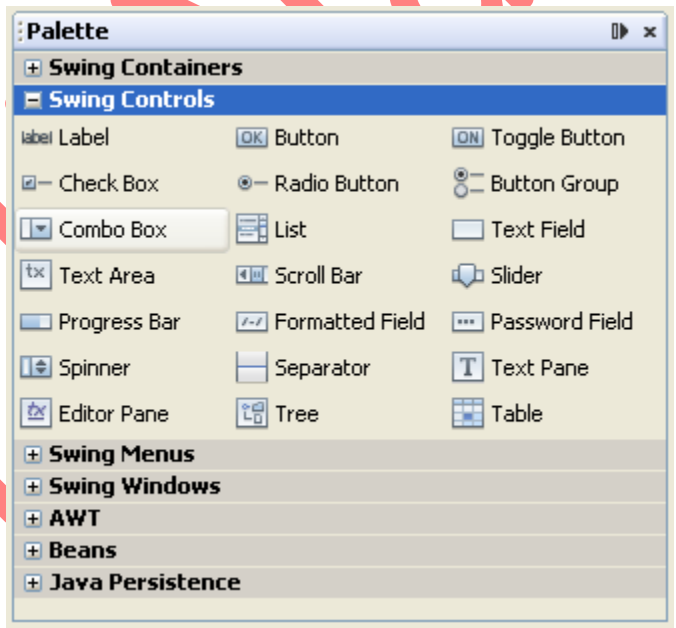


You will then have a Class called FormObjects, which is in the package called form_controls_lesson, which in the formcontrols project.

You will also have a new form on which to add controls.

The JComboBox Control

A combo box is a drop down list of items that can be selected by a user. It can be found in the NetBeans palette, under Swing Controls:

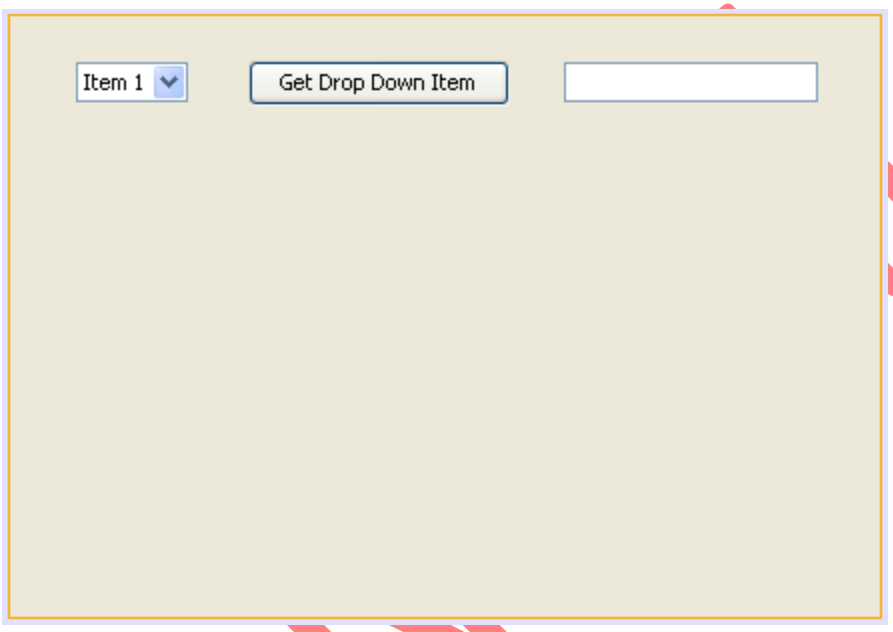


Locate the Combo Box control and drag one on to your form. Drag a Button onto the form, and a Text Field. What we'll do is to place the item selected from the drop down list into the text field. This will happen when the button is clicked.

Click back onto your Combo Box to highlight it. Right-click and select **Change Variable Name** from the menu that appears. Type **comboOne** as the new name, and then click OK.

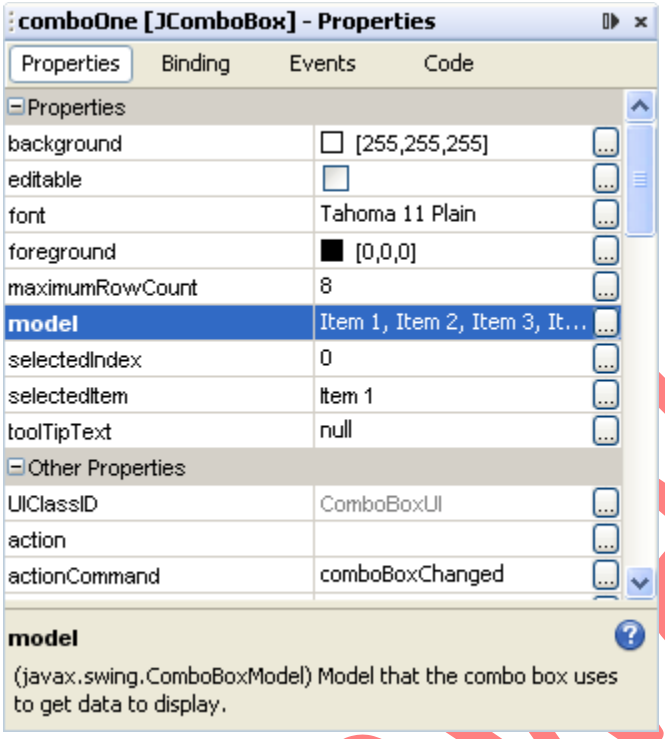
Change the name of the button in the same way, rename it **btnComboBox**. Change the text on the button to **Get Drop Down Item**.

Change the name of the Text Field to **txtComboBoxItem**. Delete the default text and leave it blank. Your form should then look something like this one:

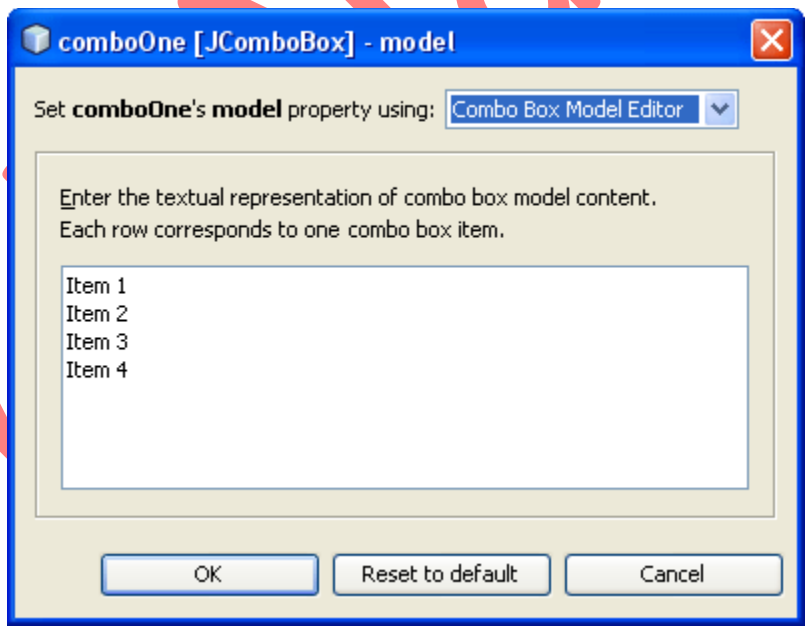


The default items in the Combo Box are **Item** 1, **Item 2**, etc. We'll add our own items.

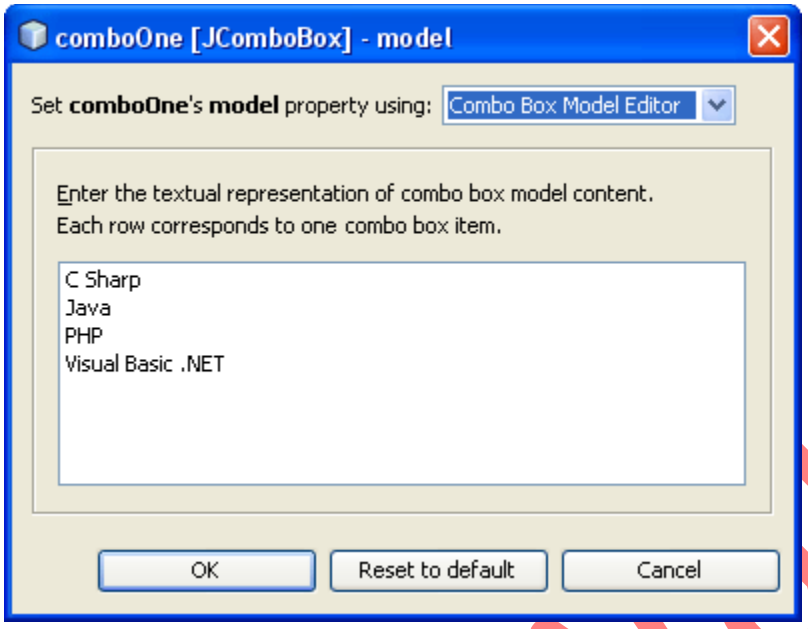
Click back onto your Combo Box to select it. Now look at the properties window on the right of NetBeans. Locate the **model** property:



Click the small button to the right of the model row, the one with the three dots in it. The following dialogue box appears:

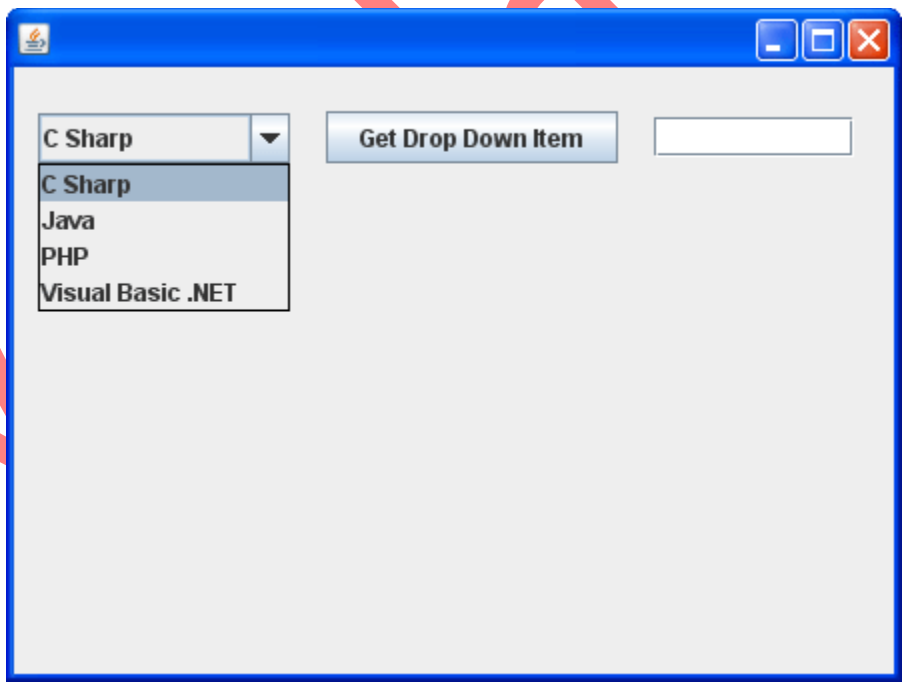


You can highlight the items in the white text area, and delete them. Replace them with the following items: C Sharp, Java, PHP, Visual Basic .NET. Your dialogue box will then look like this:



Click OK when you've made the changes. Your combo box will now be filled with your own items.

Run your programme and test it out: (Just click OK when it asks you to choose the Main Class.)



Close the programme and return to design view.

When the button is clicked, we want the item chosen to appear in the text field. So double-click your button to create a code stub.

To get which item is selected there is a handy method of combo boxes called **getSelectedItem**. But this returns an Object as a value. What we want is the text from the list. You can do something called **casting** to turn the Object into a String. Add the following line to your code stub:

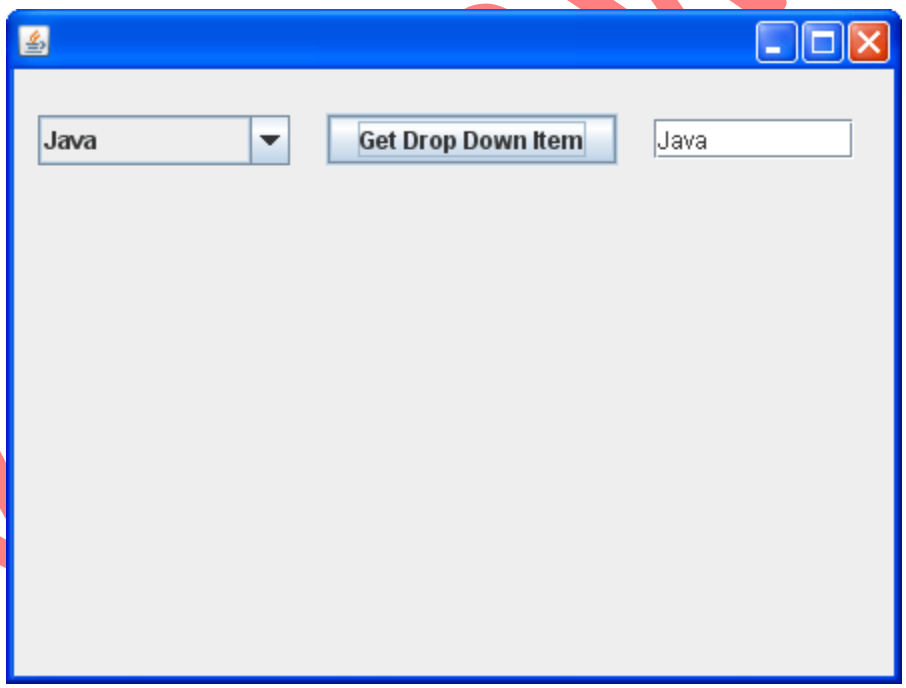
String itemText = (String)comboOne.getSelectedItem( );

So we're setting up a string variable called **itemText**. After the equals sign we use the **getSelectedItem** method of **comboOne**. But note how the casting is done - with the String variable type between round brackets. This goes immediately before the object or value you're trying to cast (casting just means converting from one variable type to another).

To display the selected item in the text field, you just need to set the text for the text field. Add this line just below the line you have:

txtComboBoxItem.setText(itemText);

Run your programme again and try it out. Select an item from your drop down list. Then click your button. The item you selected should appear in the text field:



The combo box looks a little dull, at the moment. You can it up with a bit of colour, and different fonts.

Stop your programme and return to Design view in NetBeans. Click on your combo box to select it. Now have a look at the properties window again. Try setting the following:

Background Colour
Foreground Colour
Font
Border

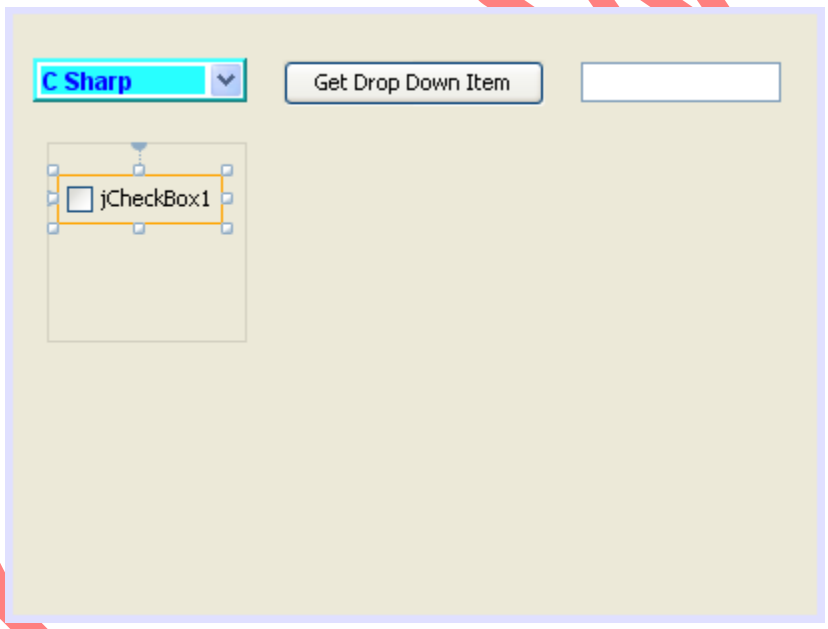
You'll need to play around with them for a while. For the colours, RGB seems to work best.

In the next lesson, you'll see how Java check boxes work.

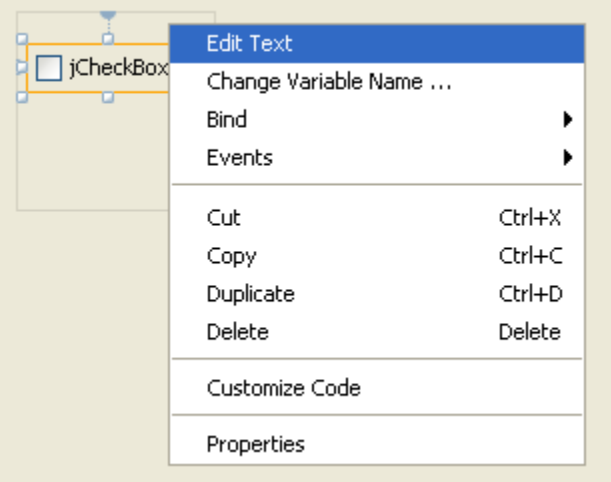
Java Check Boxes

A check box is a way to allow your users to select and deselect items. They can be a little fiddly, though, so it's a good idea to add them to a panel. That way, you can move them all at once just by moving the panel.

So add a panel to your form, which can be found under Swing Containers in the NetBeans palette. Now locate the check box control. Drag a check box onto your panel.



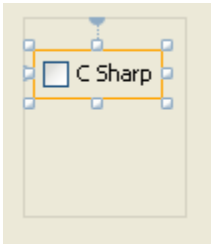
The text **jCheckBox1** is the default text. You can change this either in the properties window, or by right-clicking the check box. From the menu that appears, select Edit Text (we've chopped a few menu items off the list, in the image below):



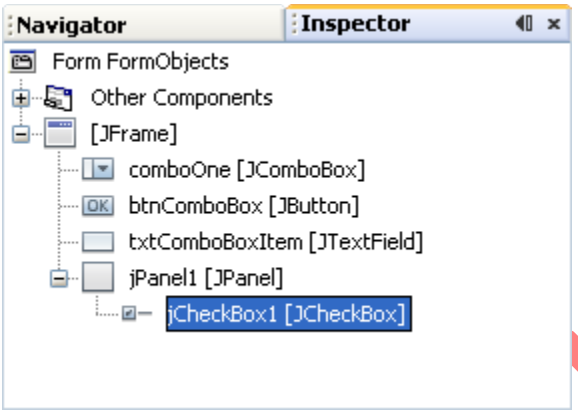
When you click on **Edit Text**, the default text will be highlighted:

Type **C Sharp** over the top of the highlighted text:

Press the enter key on your keyboard to confirm the change. The text will change for your check box:

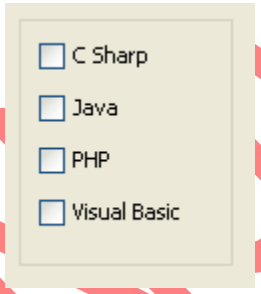


However, this just changes the text, and not the variable name. The variable name will still be **jCheckBox1**, as you can see in the **Inspector** area to the left:



Leave it on the default variable name. But just be aware that changing the text of a control does not change its variable name.

Now that you have added one check box to your panel, add three more. Change the text of the three to: Java, PHP, and Visual Basic. Your check boxes will then look like this:

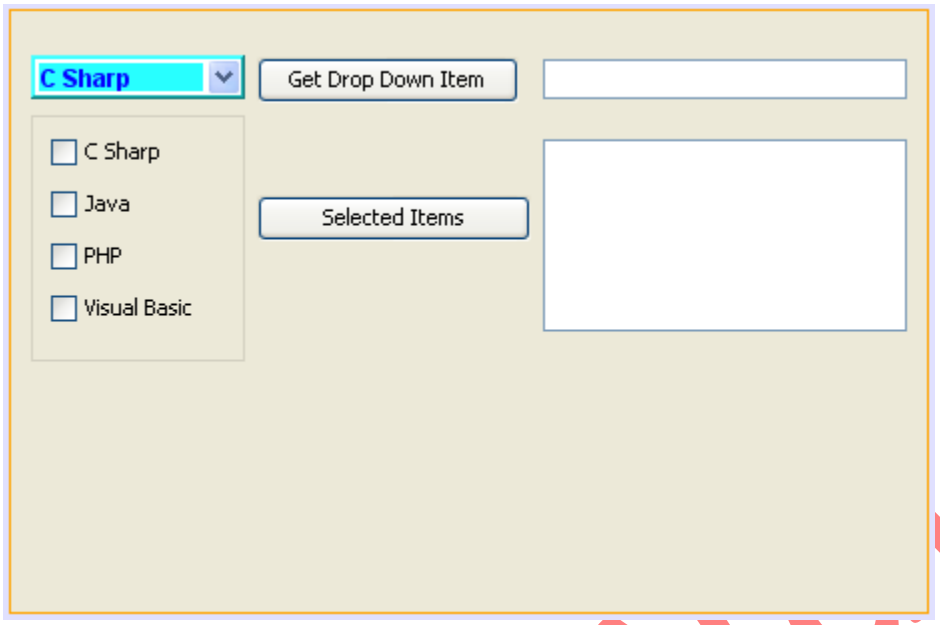


What we'll do is to get the items that a user has checked. We'll do this when a button is clicked. To display the items, we'll use the Text Area control, rather than a text field.

So add a button to your form. Change the variable name to **btnCheckBoxes**. Change the text on the button to **Selected Items**.

Locate the Text Area control in the NetBeans palette, and drag one onto your form. Change the variable name to **taOne**.

When you've aligned your new controls, your form should look something like this:



Now for the code.

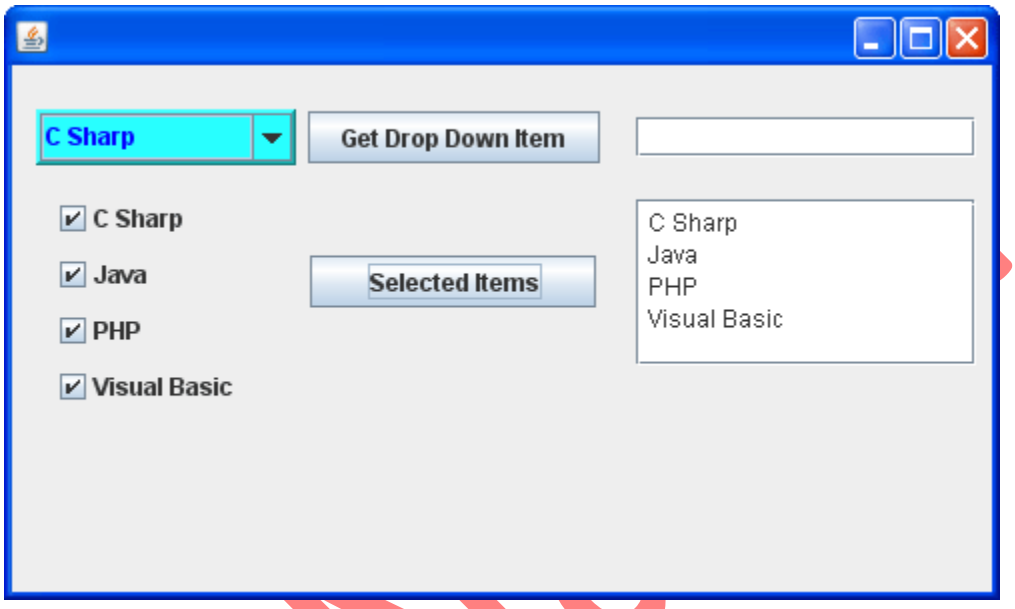
Java checkboxes have a property called **isSelected**. We can use this in a series of IF Statements to see if each box is selected or not. If they are, we can build up a string, adding the text from each check box.

Double click your new button to create a code stub. Add the following code:

```
String s1 = "";
if (jCheckBox1.isSelected()){
    s1 = s1 + " " + jCheckBox1.getText() + '\n';
}

if (jCheckBox2.isSelected()){
    s1 =  s1 + " " +  jCheckBox2.getText() + '\n';
}

if (jCheckBox3.isSelected()){
    s1 =  s1 + " " +  jCheckBox3.getText() + '\n';
}

if (jCheckBox4.isSelected()){
    s1 =  s1 + " " +  jCheckBox4.getText() + '\n';
}

taOne.setText(s1);
```

The string we're building up is called **s1**. If a check box is selected then we get the text from that check box. This is then stored in the s1 variable, along with a new line character ('\n') The last line of code sets the text for the text area. In between the round brackets of setText, we have the s1 variable, which is the string we're building up.

When you've finished typing the code, run your programme. Select a few check boxes and then click the button. You should find that the items you checked appear in the text area:

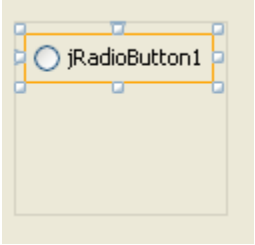


Deselect a box or two and try again. Only the boxes selected should appear in the text area.

Java Radio Buttons

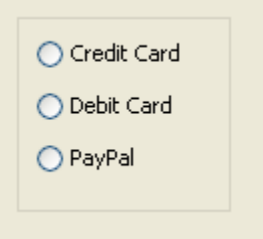
Radio buttons are usually used to select just one item from a list, rather than the multiple items available with check boxes. Let's see how they work.

Drag and drop a panel onto your form. Then locate the Radio Button control in the NetBeans palette. Drag a Radio button onto your new palette. It should look like this:

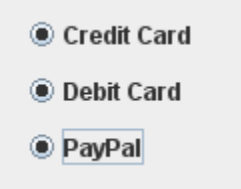


The default text for the first radio button is **jRadioButton1**. We'll use our radio buttons to allow a user to select a payment method. So change the text of your radio button to **Credit Card**. (The text can be changed in the same way as you did for check boxes. Again, we'll leave the variable name on the default of **jRadioButton1**.)

Add two more radio buttons to the panel. Change the text to Debit Card, and PayPal:



There is, however, a problem with the radio buttons you've just added. To see what the problem is, run your programme again. Now select one of the radio buttons. Try selecting another radio button and you'll find that you can indeed select more than one at the same time:



With our radio buttons, though, we only want the user to select one payment option. To solve the problem, Java lets you to create something called a **ButtonGroup**. As its name suggest, this allows you to group buttons under one name. You can then add radio buttons to the group. Once you've added buttons to the group, only one option is available for selection.

To see how the ButtonGroup works, add the following method to your code, somewhere near the top:

```
private void groupButton( ) {

ButtonGroup bg1 = new ButtonGroup( );
```

bg1.add(jRadioButton1);
bg1.add(jRadioButton2);
bg1.add(jRadioButton3);

}

When you do, you'll see that NetBeans has alerted to you to a problem, and underlined some code in red. It has done this because it can't find a class called ButtonGroup, so can't create a new object from it.

To solve this problem, you need to import the relevant class from the Swing library. So scroll up to the very top of your code, and add the following import statement:

import javax.swing.ButtonGroup;

The red underline should now be gone.

Our groupButton method adds radio buttons to the ButtonGroup object, with the use of the add method:

bg1.add(radio_button_name);

There's one line for every radio button on our form.

We can call the groupButton method from the constructor. That way, the radio buttons will be grouped when the form loads. Add the following method call to your constructor:

```
/** Creates new form FormObjects */
public FormObjects() {
    initComponents();
    groupButton();
}
```

The top of your code window should look like this:

```
package form_controls_lesson;

import javax.swing.ButtonGroup;

public class FormObjects extends javax.swing.JFrame {

    /** Creates new form FormObjects */
    public FormObjects() {
        initComponents();
        groupButton();
    }
```

Run your form again, and try to select more than one radio button. You should find that you can only select one in the group.

To get at which radio button was selected, again there's an **isSelected** method we can use.

Add a normal button to your form. When we click this button we'll display a message box stating which radio button was clicked.

Change the variable name of your button to btnRadios. Change the text property to Payment Option.

Now double click your new button to create a code stub. Add the following:

```
String radioText = "";

if (jRadioButton1.isSelected()) {
    radioText = jRadioButton1.getText();
}

if (jRadioButton2.isSelected()) {
    radioText = jRadioButton2.getText();
}

if (jRadioButton3.isSelected()) {
    radioText = jRadioButton3.getText();
}
```

All we're doing here is checking which radio button is selected. We're then getting the text from the radio button and storing it in a variable called radioText.

We can have a message box to display which payment option was selected. Add the following line to the bottom of your button code, just below the final IF statement:

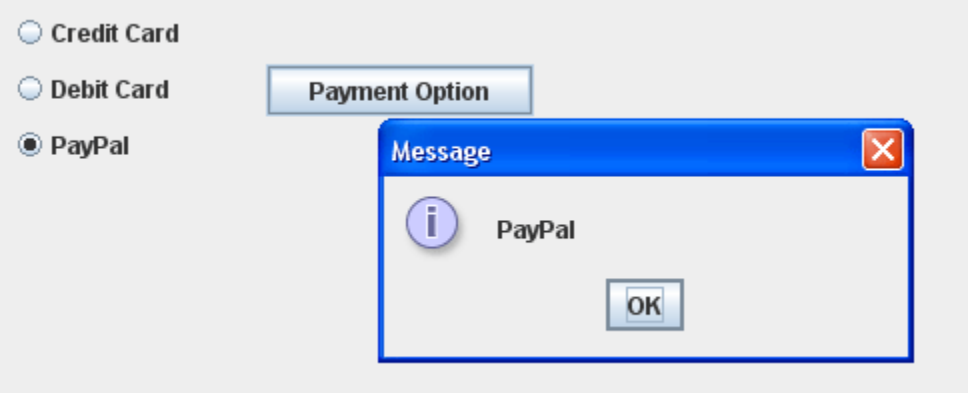
javax.swing.JOptionPane.showMessageDialog(FormObjects.this, radioText);

This line is so long that we've had to reduce the font size! But you've met the JOptionPane in a previous section. The only difference is the first item between the round brackets. Because we were using a console, the first item was null. Now we have:

FormObjects.this

The first item between the round brackets is for the window in which you want to display the message box. **Null** meant no window. **FormObjects.this** means this component (the form) of the FormObjects class.

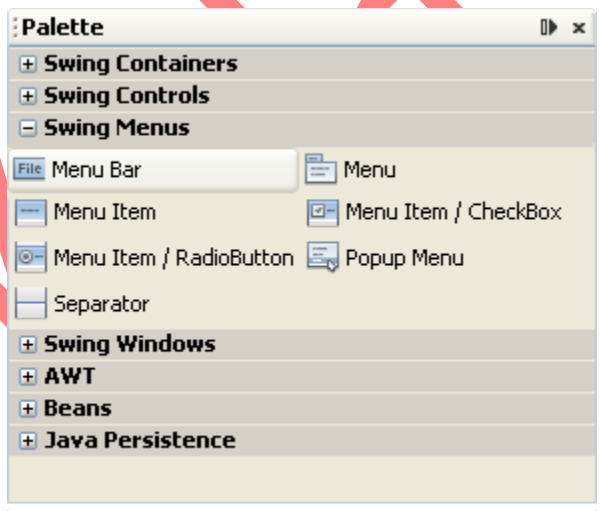
Run your programme again, and select an item from your radio button. Then click your button. You should see something like the following:



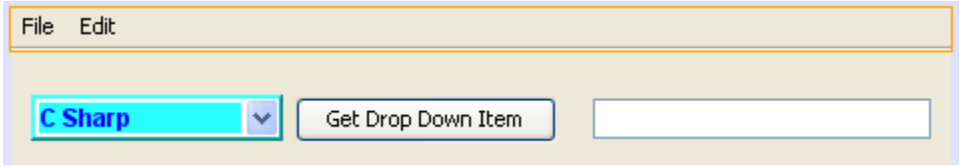
Java Menus

You can add menus to your Java forms, things like File, Edit, View, etc. Each menu has menu items, and these in turn can have sub menus.

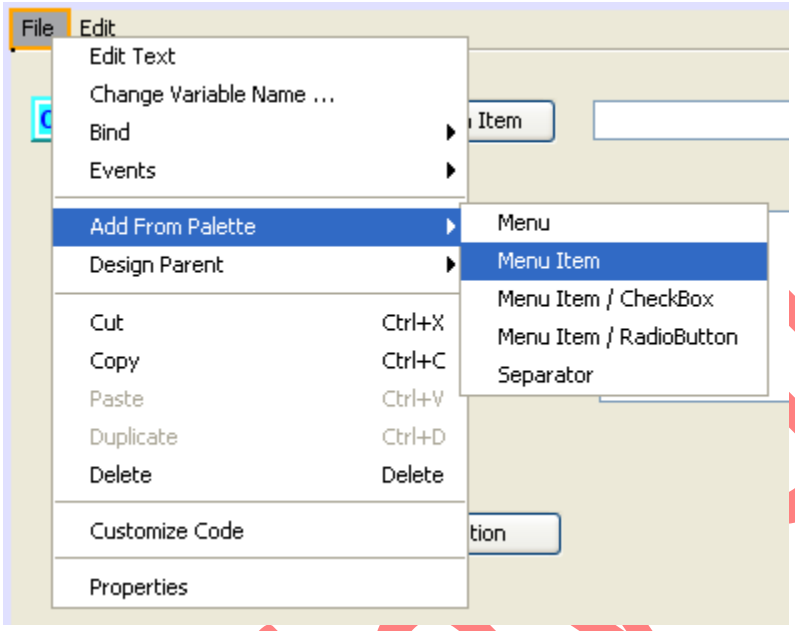
Return to Design view. In the NetBeans palette, locate the Menu Bar item:



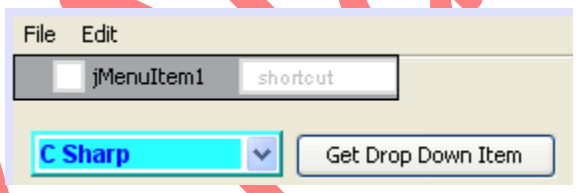
Drag one to the top of your form. When you let the mouse button go, you'll have a default File and Edit menu bar:



There's no menu items added by default, though. To add your own, click on the **File** menu item to select it. With the File menu item selected, right-click. A new menu will appear. Select **Add From Palette > Menu Item**:

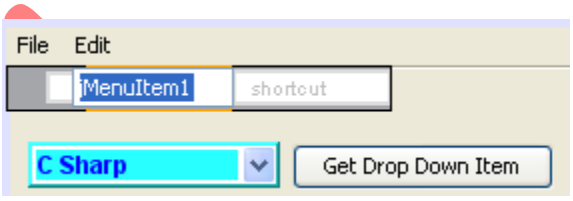


A Menu Item will be added to your File menu:

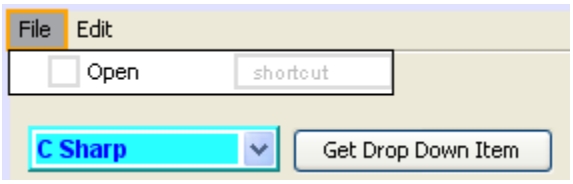


What we'll do is to add menu items to open and save a file.

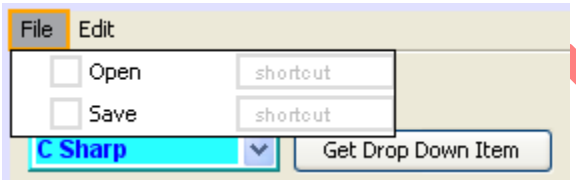
Double click on the default text **jMenuItem1**. It will then be highlighted, so that you can type over it:



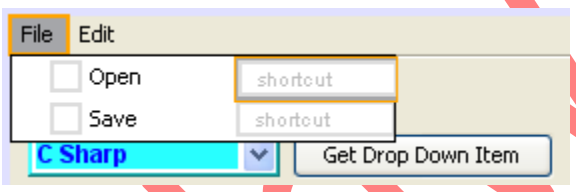
Type **Open**, then press the enter key on your keyboard:



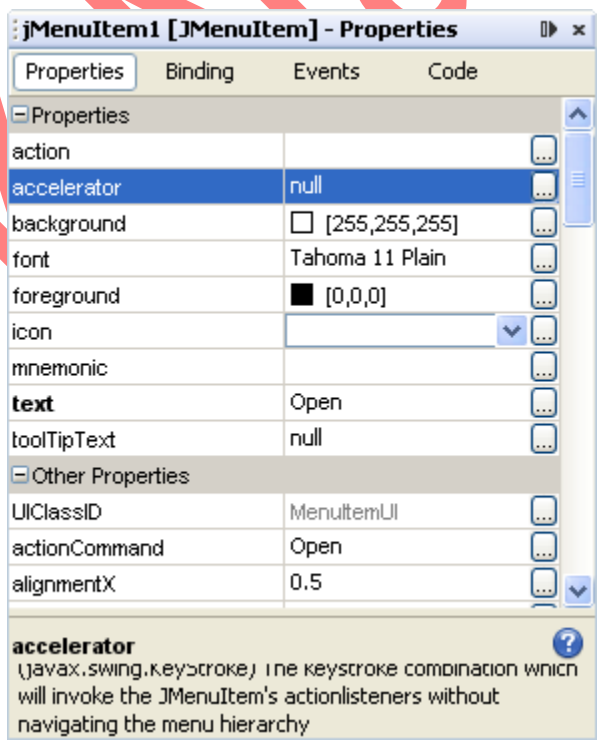
Add another menu item in the way. This time, type **Save** as the menu item:



As you can see above, you can add shortcuts for your menu items. Click on to the Open menu item, then onto the shortcut for it:

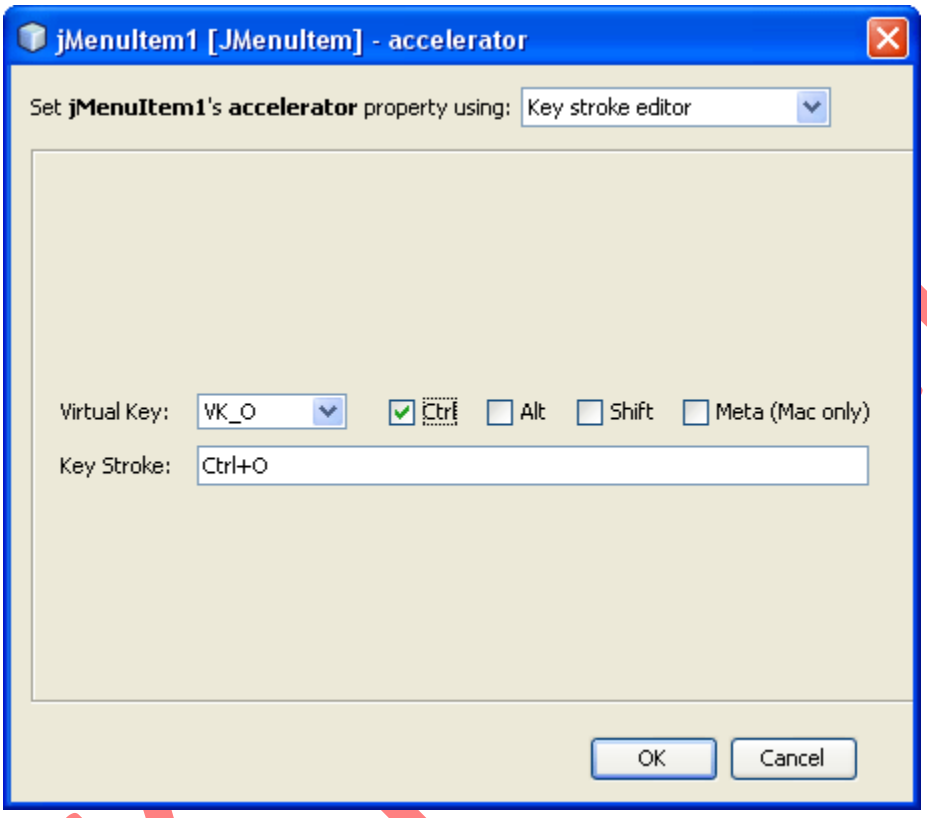


With the shortcut item selected, have a look at the properties window:

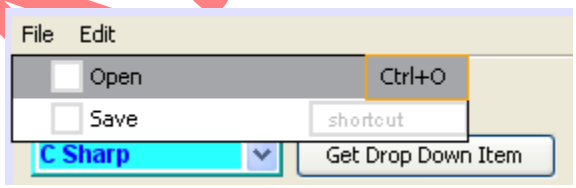


Locate the **Accelerator** item, and click the small button to the right of the row. A dialogue box appears. You can set which shortcut keys you want for a menu item from this dialogue box. An open shortcut is usually CTRL + O.

Type an O in the box, and Shift + O will appear. Uncheck the Shift item and check Ctrl instead:



Click OK, and the shortcut will be added to your Java menu item:



To see if all this works, click back on the Open menu item to highlight it. Now right click. From the menu that appears, select **Events > Action > Action Performed**. This will create a code stub for the menu item. Enter the following for the code:

```
javax.swing.JOptionPane.showMessageDialog( FormObjects.this, "Open" );
```

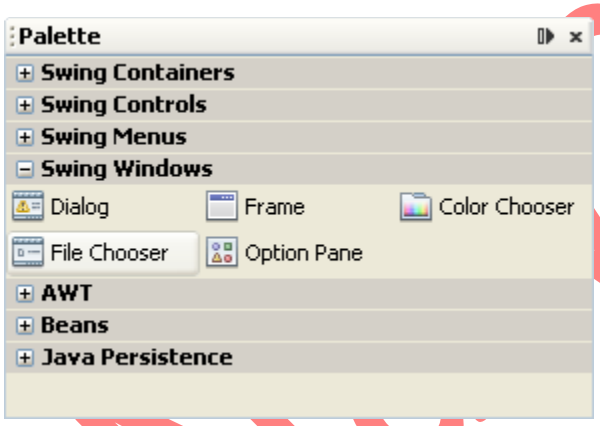
This is just a message box.

Run your programme and try it out. Click **File > Open** and you should see the message box appear. Click OK to get rid of it. Now try your shortcut. Hold down the Ctrl key on your keyboard. Keep it held down and press the letter O. Again, the menu should appear.

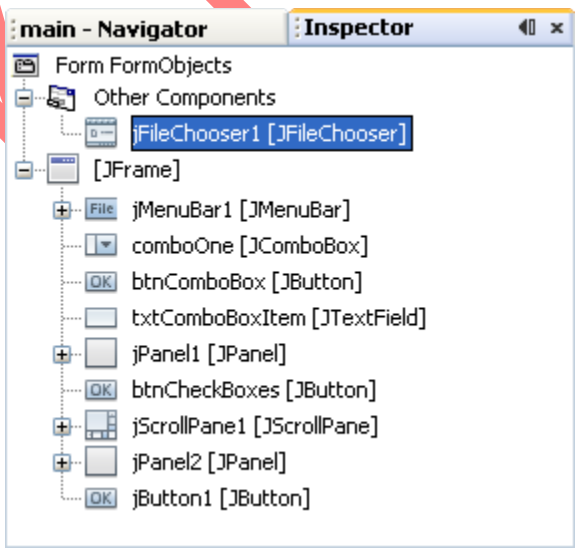
The Java File Choser

In this lesson, you'll see how to display Open File dialogue boxes in Java. This is done with the **File Choser** control.

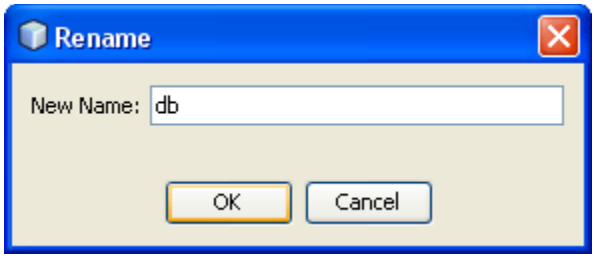
Go back to Design view. In the NetBeans palette, locate the File Chooser item, which is under Swing Windows:



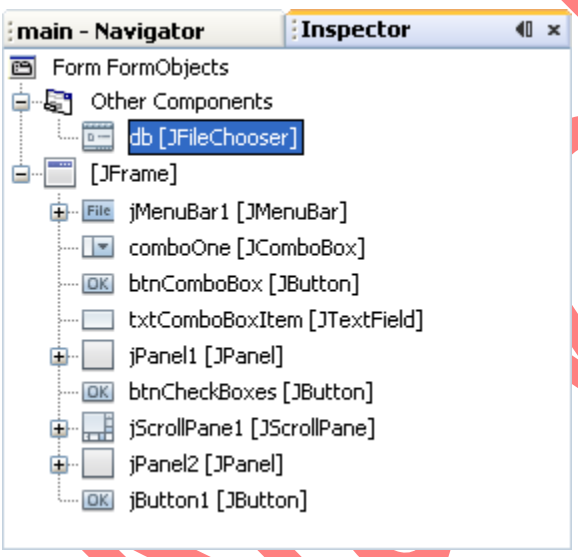
Drag a File Chooser near your form, but not onto it. Drop it just below the form, in a white area. It won't actually appear on the form, but you can see it in the Inspector window:



The default name for the File Chooser is **jFileChooser1**. Right click on jFileChooser1 in the Inspector window. From the menu that appears, select Change Variable Name. When the dialogue box appears, type db as the name:



Click OK to confirm the change. The Inspector window should look like this:



You now have a File Chooser added to the project.

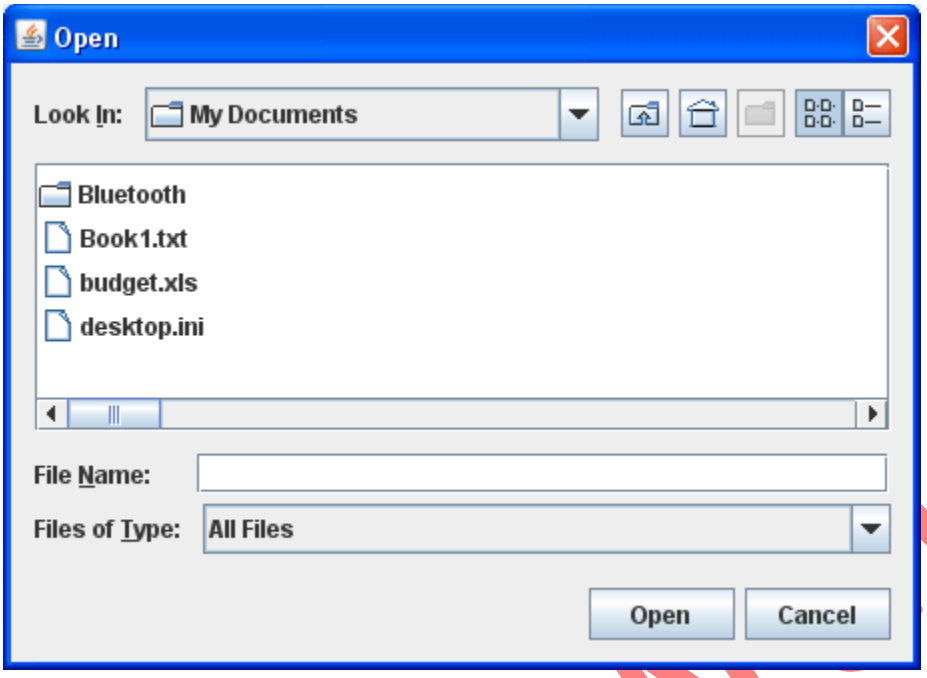
Displaying the File Chooser dialogue box is quite simple. Go back to the code stub for your Open menu item, the one where you had the message box. Now type the following line:

int returnVal = db.showOpenDialog(this);

Our File Chooser is called **db**. We're using the **ShowOpenDialog** method of the File Chooser class. In between the round brackets of ShowOpenDialog you type then name of the window that's going to be holding the dialogue box. We've typed this, meaning "this form".

The ShowOpenDialog method returns a value. This value is an integer. The value tells you which button was clicked on the dialogue box: open, cancel, etc. We're storing the value in a variable called returnVal. We'll use this value in a moment.

But run your programme again. Click **File > Open** on your form. You should see a dialogue box appear:



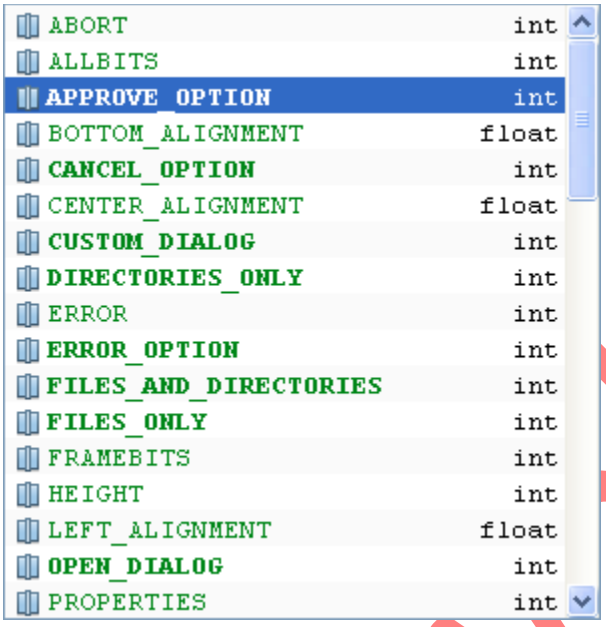
Unfortunately, the only thing an Open Dialogue box does is to select a file - it doesn't actually open anything. If you want to open a file, you have to write the code yourself. We'll do that soon. But for now, we can get the name and path of the file that a user selects.

First, add the following IF Statement to your code, just below your other two lines (the returnVal one and your commented-out message box):

if (returnVal = = javax.swing.JFileChooser.APPROVE_OPTION) {

}

So we're using a Swing class called JFileChooser. With this class, you can examine which button was clicked. When you type the dot after JFileChooser you should see a popup list:



The APPROVE_OPTION means things like the OK and Yes buttons. So we're testing the **returnVal** variable to see if it matches the APPROVE_OPTION (did the user click OK?).

To get at the file chosen by the user there is a method called getSelectedFile. However, this returns a File object, rather than a string. The File object is part of the **IO** class in Java. So add the following line to your IF Statement:

java.io.File file = db.getSelectedFile( );

So the file chosen by the user will end up in the File object that we've called file.

To do something useful with it (open the file, for example), we need to convert it to a string:

String file_name = file.toString( );

This line just uses the **toString** method of File objects. We're placing the result in a new variable called **file_name**. Add the line to your IF Statement.

To display the file name, remove the comments from your message box and move it as the last line of your IF Statement. Change the last parameter between the round brackets to file_name:

javax.swing.JOptionPane.showMessageDialog(FormObjects.this, file_name);

Because this line is so long, you can add an import statement to the top of your code, underneath the one you already have:

import javax.swing.JOptionPane;

The message box line can then just be:

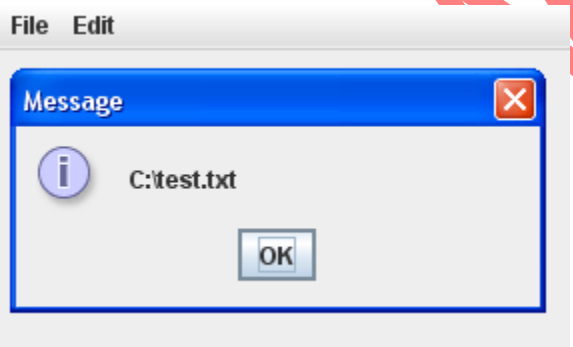
JOptionPane.showMessageDialog(FormObjects.this, file_name);

It's a little bit easier to read, now!

But your code should now look like this:

```
int returnVal = db.showOpenDialog(this);

if (returnVal == javax.swing.JFileChooser.APPROVE_OPTION) {
 java.io.File file = db.getSelectedFile();
 String file_name = file.toString();
 JOptionPane.showMessageDialog(FormObjects.this, file_name);
}
```

Run your programme and test it out. Click **File > Open** to see the dialogue box. Select any file on your computer and click **Open**. Your message box should display:



Before we add the code to open up the selected file, you may have noticed that **Files of Type** on your Open file dialogue box is set to "All files". You can filter the files on this list, so that the user can open only, say, text files, or just images with certain extensions (jpeg, gif, png).

To filter the "Files of Type" list, the dialogue box has an **addChoosableFileFilter** method. But you need a Filter object between the round brackets.

So that your code doesn't get unnecessarily long, add the following two import statements to the top of your code, just below the other ones:

import javax.swing.filechooser.FileFilter;
import javax.swing.filechooser.FileNameExtensionFilter;

To set up a file name extension filter, you need to create a new FileFilter object. Add the following line just before the first line of your code (before the int returnVal line):

FileFilter ft = new FileNameExtensionFilter("Text Files", "txt");

In between the round brackets of **FileNameExtensionFilter** you first need the text that will appear on the **Files of Type** list. After a comma you type the name of files that you want to display. A proper file extension is needed, here, but without the dot. Note the double quotes above.

You can add more than one extension. Just type a comma and then the files you want to display:

FileNameExtensionFilter("Text Files", "txt", "html");

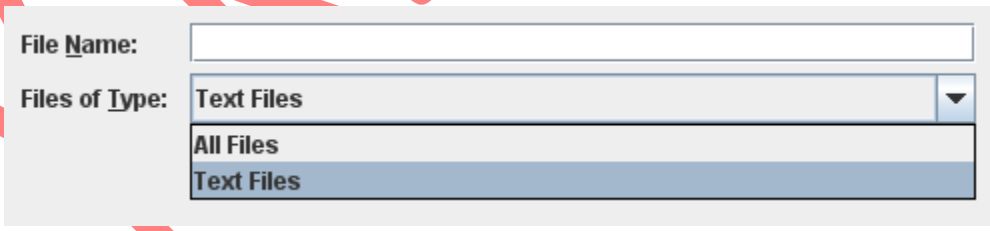
Once you have a filter object set up, you can use the **addChoosableFileFilter** method of your dialogue box:

db.addChoosableFileFilter(ft);

Add the line to your code, just below the FileFilter line:

```
FileFilter ft = new FileNameExtensionFilter("Text Files", "txt");
db.addChoosableFileFilter(ft);
int returnVal = db.showOpenDialog(this);

if (returnVal == javax.swing.JFileChooser.APPROVE_OPTION) {
  java.io.File file = db.getSelectedFile();
  String file_name = file.toString();
  JOptionPane.showMessageDialog(FormObjects.this, file_name);
}
```

Run your programme again, and have a look at your dialogue box. Click the arrow on the dropdown list:

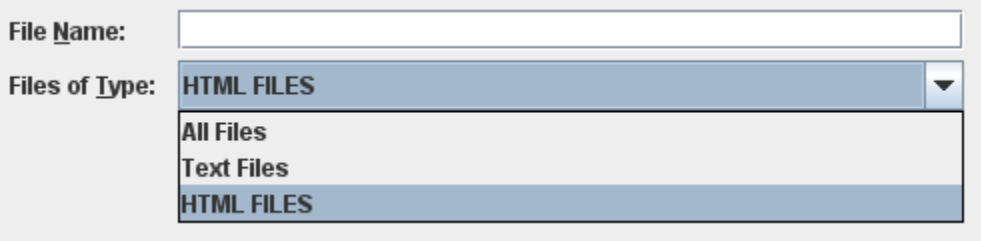


Select the Text Files option. Your dialogue box will then only displays files with the extension .txt.

If you want another line on the list, (to display html files, for example), you can set up another FileFilter object:

```
FileFilter ft = new FileNameExtensionFilter("Text Files", "txt");
FileFilter ft2 = new FileNameExtensionFilter("HTML FILES", "html");

db.addChoosableFileFilter(ft);
db.addChoosableFileFilter(ft2);
```

When the programme is run, the Files of Type list would then look like this:

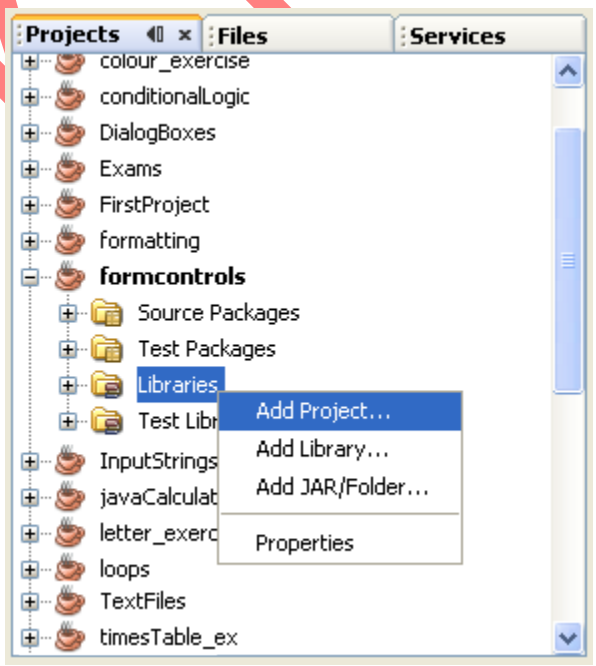


Using a Dialogue Box to Open Files in Java

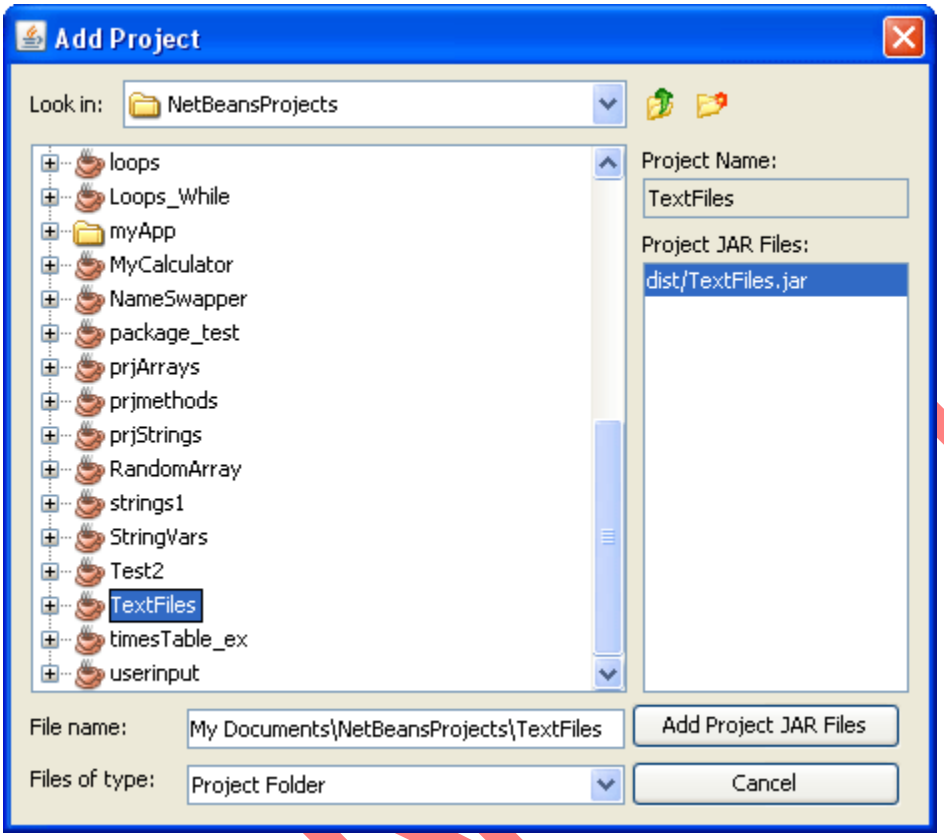
Actually, you have already written code to open a file. The process is no different for the open file dialogue box. And because you have already written the classes that open and write to a file, you can just import them into the current project.

To import a class that you have already written, have a look at the Properties window on the left of NetBeans. (If you can't see the properties window, click the **Window** menu item at the top of NetBeans. From the Window menu, select **Properties**.)

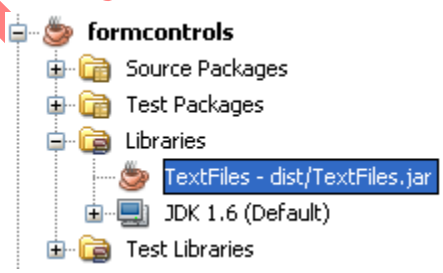
Expand the entry for your current project, and right-click the Libraries item:



From the menu that appears, select Add Project. When you do, you'll see the following dialogue box:



Make sure that the Look in box at the top says "NetBeans Projects". Now scroll down and locate your TextFiles project. This is where your classes are that open and write to a file. Now click the "Add project JAR files" button. The dialogue box will close and you'll be returned to NetBeans. Have a look at the Libraries entry for your current project and you'll see that it has been imported:



Now that you've added the classes to your project, you can add an import line to the top of your code:

import textfiles.ReadFile;

If your recall, ReadFile was what we called the class that opened a file. The package name we came up with was called textfiles. We're now importing the ReadFile class, and can now create new objects from it.

ReadFile file_read = new ReadFile(file_name);

Adapt your IF Statement to add the following **try …catch** code:

```
if (returnVal == javax.swing.JFileChooser.APPROVE_OPTION) {

    java.io.File file = db.getSelectedFile();
    String file_name = file.toString();

    try {
        ReadFile file_read = new ReadFile(file_name);
        String[] aryLines = file_read.OpenFile();
        int i;
        String theText = "";
        for (i=0; i < aryLines.length; i++) {
            theText = theText + aryLines[i] + '\n';
        }

        taOne.setText(theText);
    }
    catch (java.io.IOException e) {

    }
}
```

The code is just about the same as when we opened a text file previously: get the name of the file and create a new **ReadFile** object, call the **openFile** method we created, and return all the lines as an array, then loop round reading each line. Notice that we place all the lines in the text area control on our form.

Exercise

One thing missing, however, is a message in the **catch** part of the **try … catch** block. Move your message box between the curly brackets of catch. In between the round brackets of **showMessageDialog** add a suitable message. Or you can simply use **e.getMessage**.

When you've completed the code, try it out. You should be to open a file using your File > Open menu. The text file should then appear in your text area field.

The Save File Dialogue Box

If you want to display the Save File dialogue box instead of the Open File dialogue box then you can use your File Chooser again. This time, instead of showOpenDialog you use **showSaveDialog**:

int returnVal = db.showSaveDialog(this);

Again, this is enough to display the dialogue box. You don't have to do anything else, though you can add a file filter, as well.

FileFilter ft = new FileNameExtensionFilter("Text Files", "txt");
db.addChoosableFileFilter(ft);

int returnVal = db.showSaveDialog(this);

To write the file, add an import statement to the top of your code:

import textfiles.WriteFile;

You can then create a new WriteFile object. Again, you can place the code inside of an IF Statement, along with a **try … catch** block:

```
FileFilter ft = new FileNameExtensionFilter("Text Files", "txt");
db.addChoosableFileFilter(ft);
int returnVal = db.showSaveDialog(this);

if (returnVal == javax.swing.JFileChooser.APPROVE_OPTION) {

        java.io.File saved_file = db.getSelectedFile();
        String file_name = saved_file.toString();
    try {
        WriteFile data = new WriteFile(file_name, false);
        String alltext = taOne.getText();
        data.writeToFile(alltext);
    }
    catch (java.io.IOException e) {
        //YOUR_MESSAGE_BOX_HERE
    }
}
```

The last two lines in the code above get the data from the text area. This is then placed into a string variable called alltext. We then call the writeToFile method from our WriteFile class.

Exercise

Create a code stub for your **File > Save** menu item. Add the above code to it. Run your programme and try it out. Type some new text into the text area. Then click **File > Save** on your menu. When the "Save File" dialogue box appears, type a file name and then click Save. Check your file to see if it has indeed been created, and that it contains the text you wrote in the text area.

Java and Databases

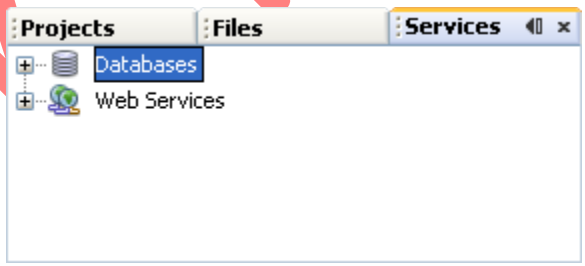
In this section, you'll learn about Java databases. You'll create a simple database with one table, and learn how to connect to it using Java code.

About Java and Databases

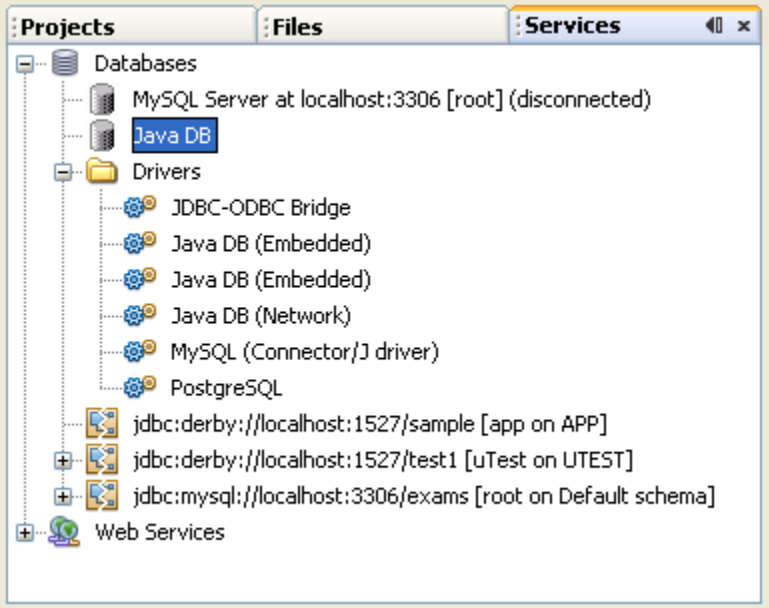
Java uses something called JDBC (Java Database Connectivity) to connect to databases. There's a JDBC API, which is the programming part, and a JDBC Driver Manager, which your programmes use to connect to the database.

JDBC allows you to connect to a wide-range of databases (Oracle, MySQL, etc), but we're going to use the in-built database you get with the Java/NetBeans software. The database is called Java DB, a version of Apache Derby. It runs on a virtual server, which you can stop and start from within NetBeans.

To check that have everything you need, have a look at the **Services** tab in NetBeans. If you can't see the Services tab, click Window from the NetBeans menu. From the Window menu, select **Services**. You should see something like this:



Expand the Databases item to see a Java DB item, and a Drivers section:



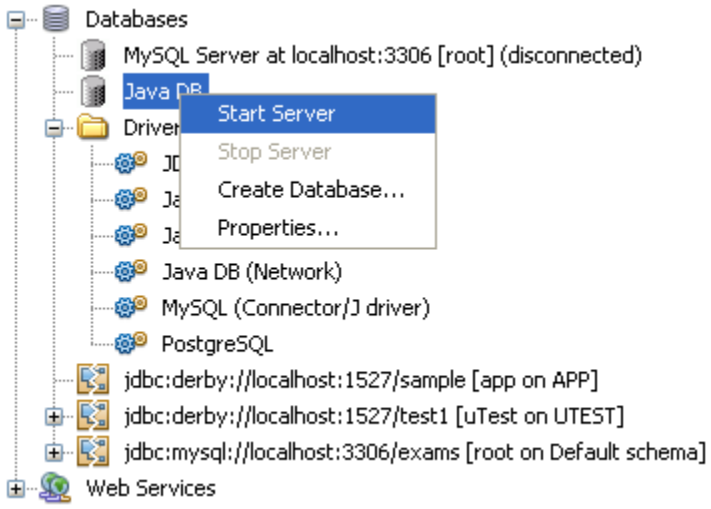
The idea is that you start the Java DB virtual server, and then create and manipulate databases on the server. There should be a database called **sample** already set up: (But don't worry if it's not there as we'll create our own database.)

In the image above, there are three databases: one is called **sample**, one is called **test1**, and the other is called **exams**.

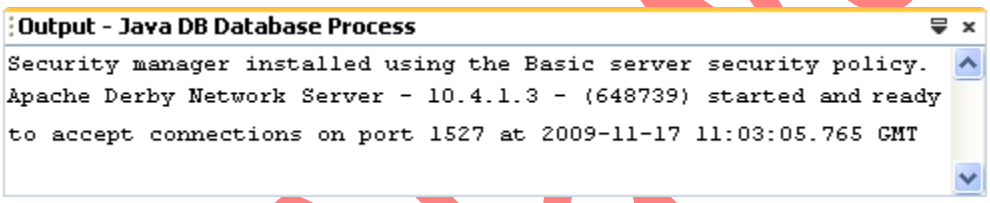
For the project in this section, we're going to set up a new database. You'll then learn how to connect to this database using Java code. The database we'll create will be a simple one-table affair, rather than multiple tables connected together. You can indeed create multiple tables with Java DB, but we don't want to complicate things unnecessarily.

Starting the Virtual Server

The first thing to do is to start the server. So right click on Java DB. You'll see a menu appear. Select **Start Server**:



Have a look at the Output window and you'll see a few messages appear: (If you have a firewall running, you'll need to let the Java DB server through.)

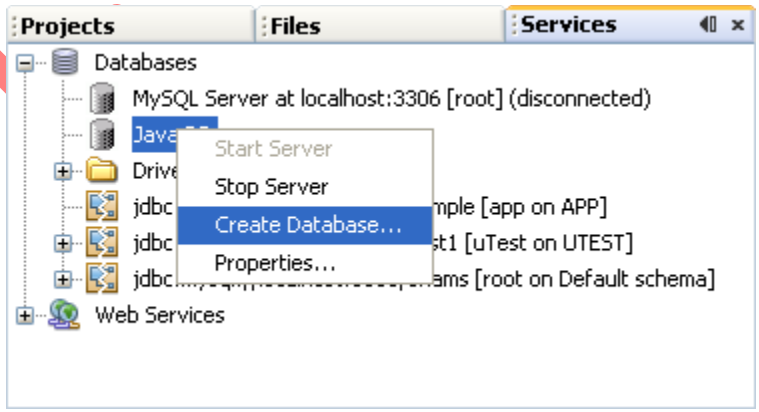


Once the server is up and running you can create databases.

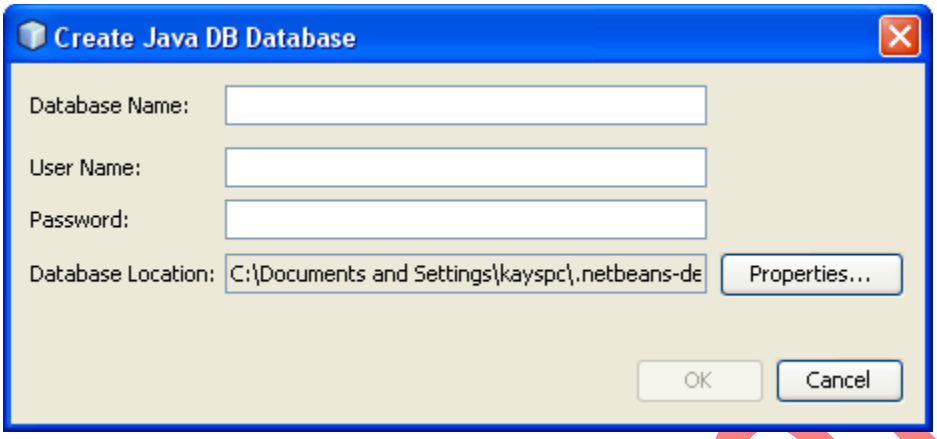
Creating a Database with Java

Now that your server has been started, you can go ahead and create a database.

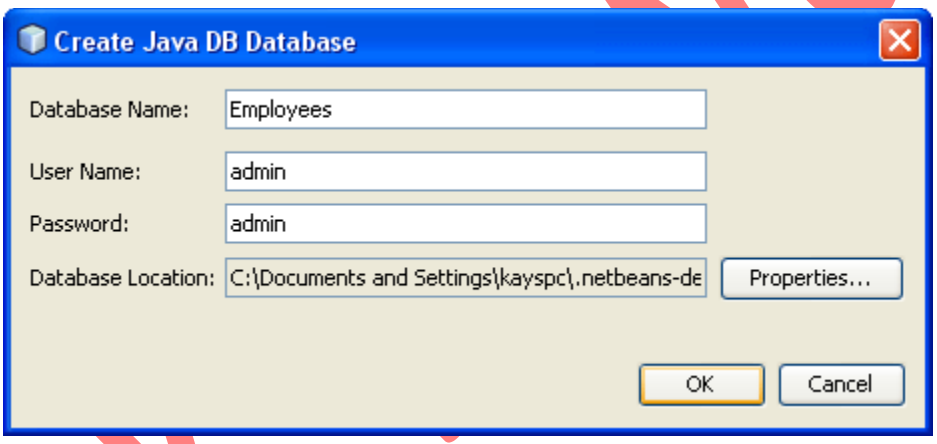
To create a new database, right click on Java DB again. From the menu that appears, select **Create Database**:



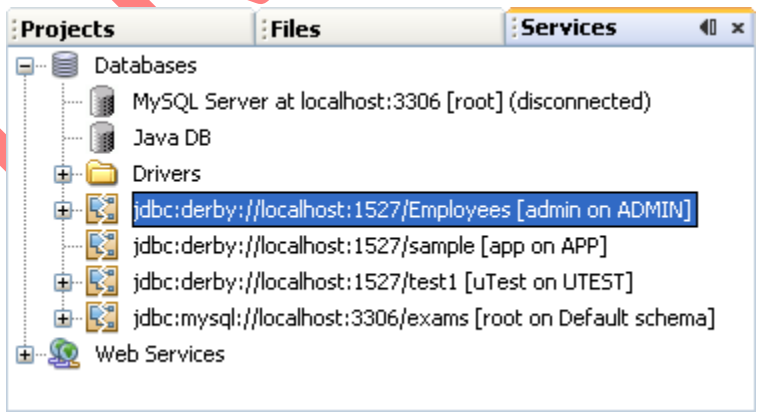
When you click on Create Database, you'll see a dialogue box appear:



Type a name for your database in the first box. Call it **Employees**. Type any User Name and Password (something a bit harder to crack than ours below!):

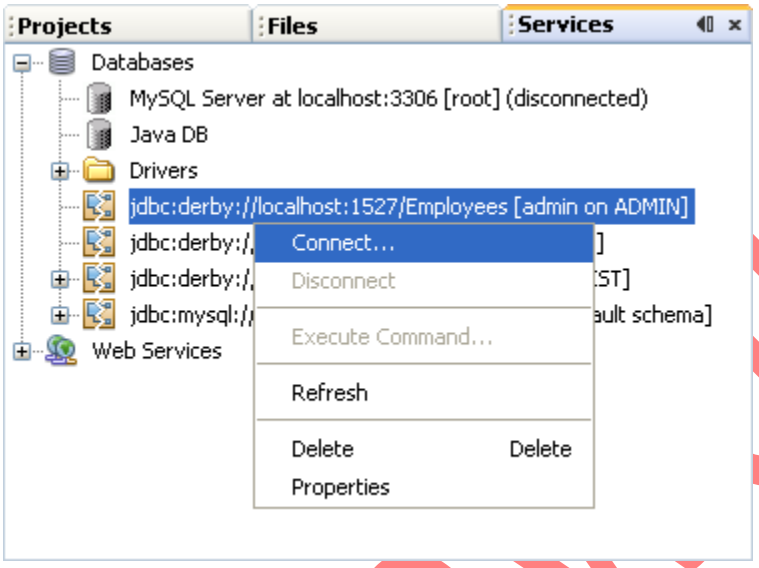


Click OK to create your database. It should then appear on the list:

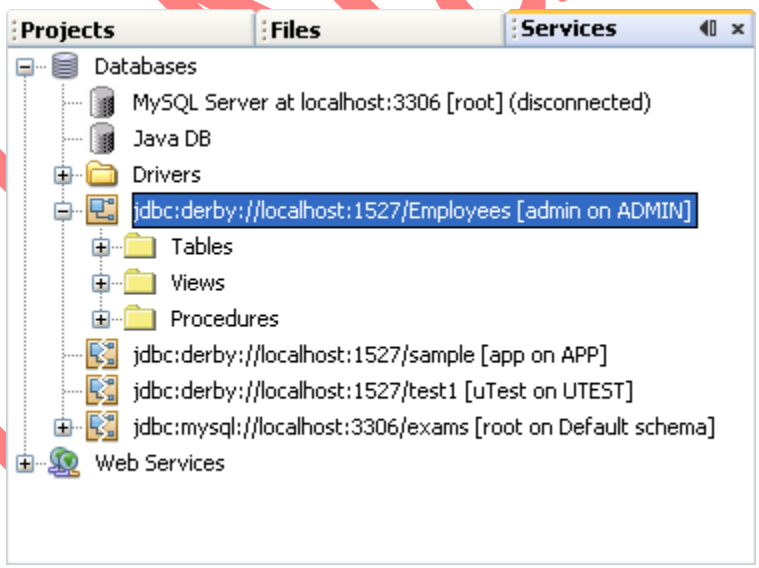


Creating a Table in the Database

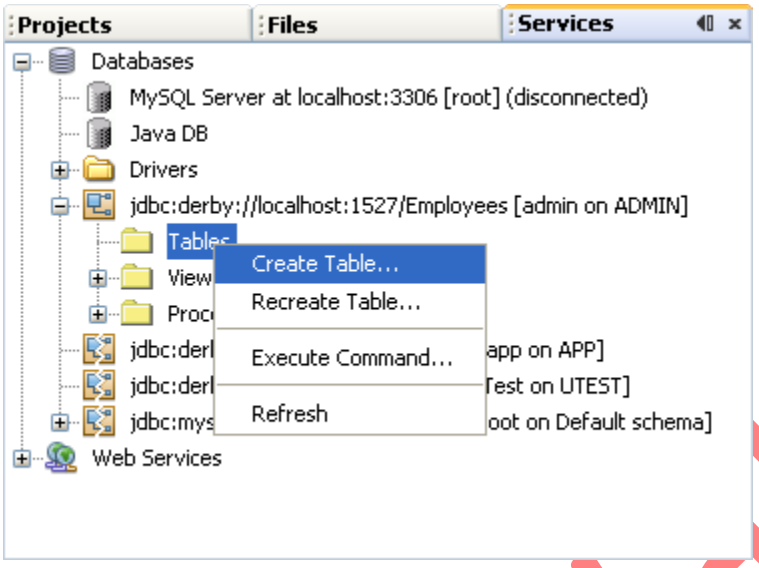
Now that the database has been created, you need to create a table in the database. To do so, right click on your database. From the menu that appears select **Connect**:



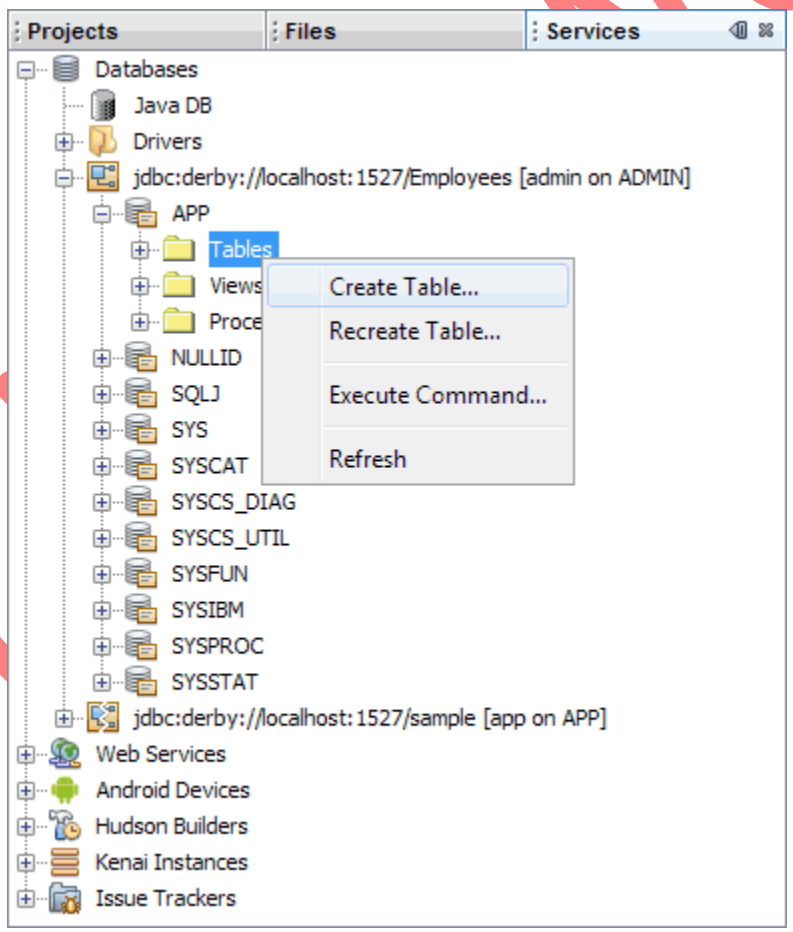
When a connection is made, you'll see some default folders for Tables, Views, and Procedures (see further down if your screen is not like this):



To create a new table in your database, right click the **Tables** folder. From the menu that appears, select **Create Table**:

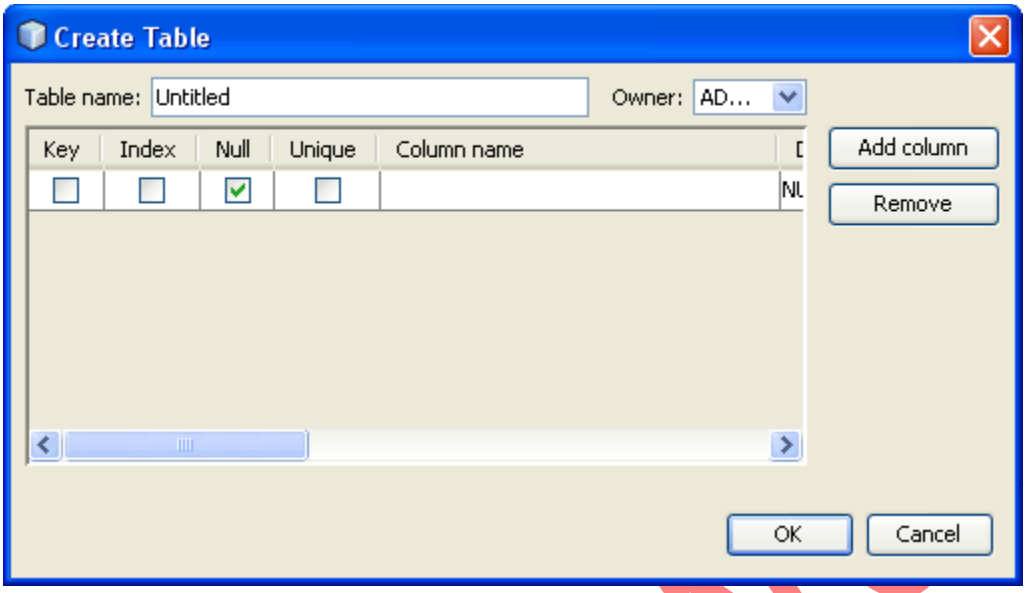


If you don't see just the three folders above, but have something like this instead:

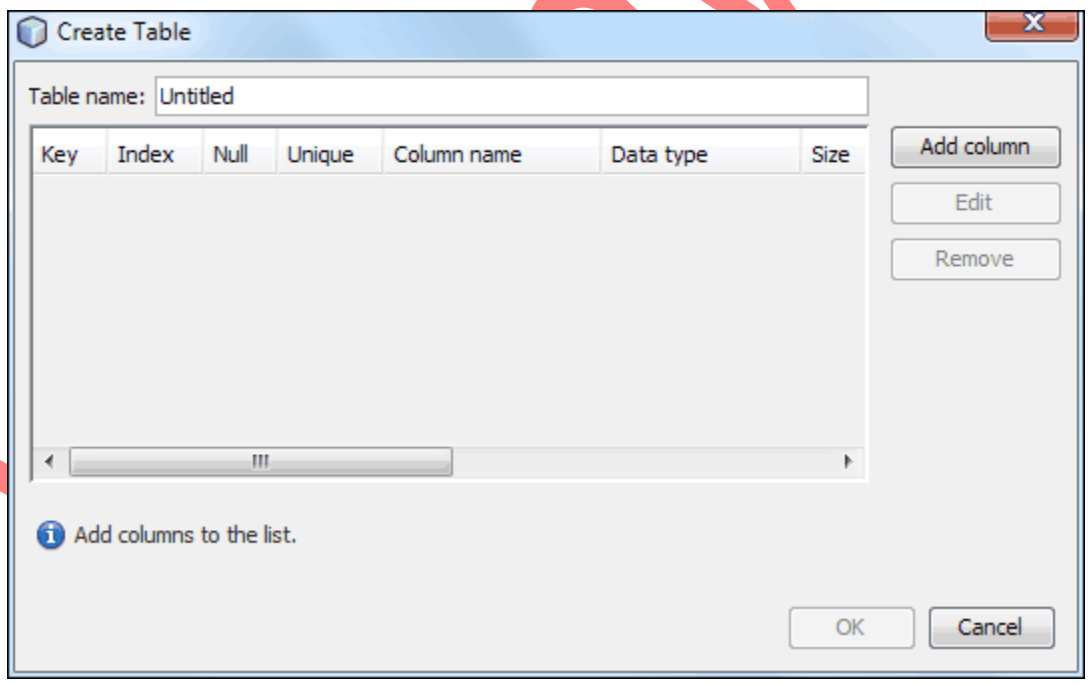


Click the APP entry, and then right-click on Tables.

When you click on Create Table, a dialogue box appears. Either this one:



Or this one:



From here, you not only type a name for your table, but you also set up the columns for the table.

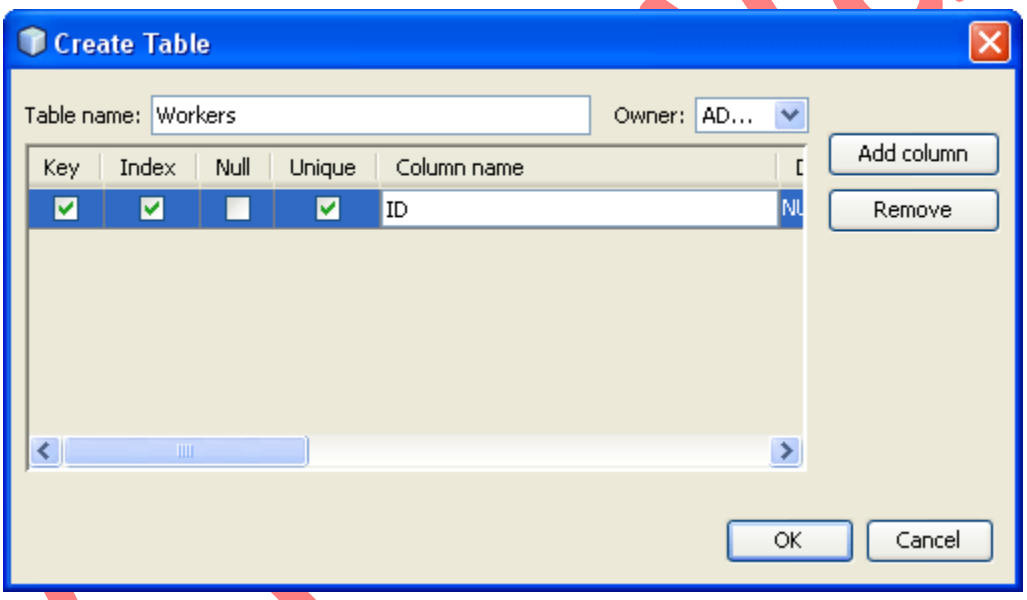
In the Table Name at the top, delete the default name of Untitled. Type a new name for your table. Call it **Workers**. You'll then have a table called Workers, which is in the **Employees** database.

But you can't click OK just yet as the table has no columns in it. We want to create columns with the following names:

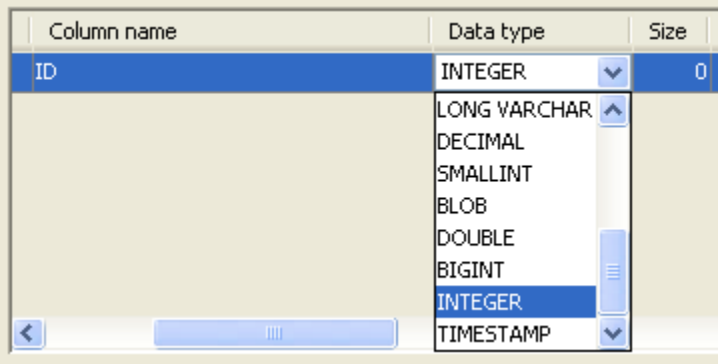
ID
First_Name
Last_Name
Job_Title

The ID column will hold a unique identifying number. This will identify a row in the table. A column with unique data in it is known as a **Primary Ke**y. Because it's the Primary Key, the column has to hold data: It can't hold a null value. (A null value just means there's no information there.)

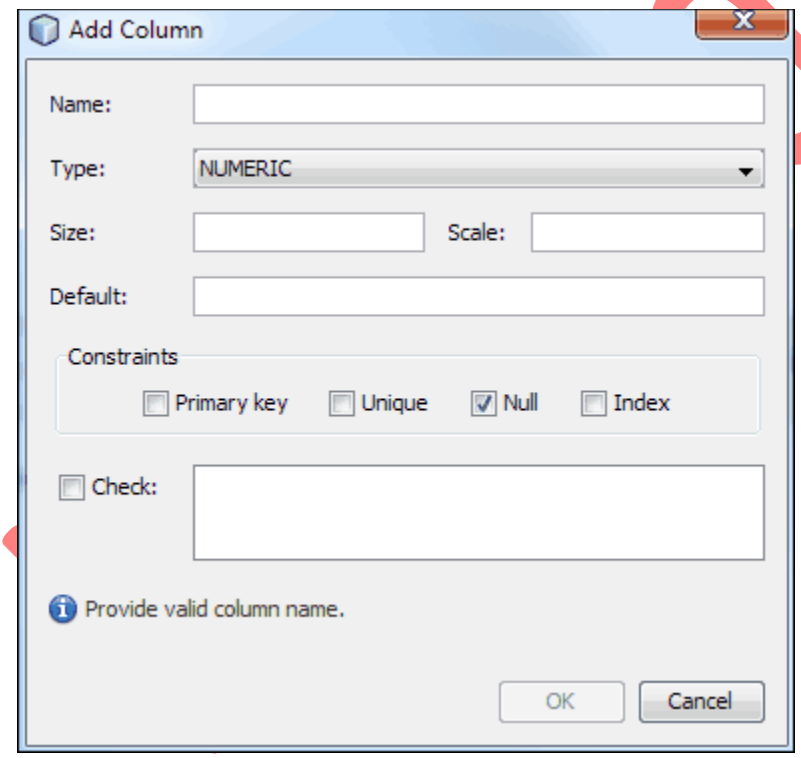
If your Create Table dialogue box is like the first one, then put a tick in the box for **Key**. When you tick the Key box, check marks will also appear for **Index** and **Unique**. Now enter a title in the Column Name area. Type **ID**:



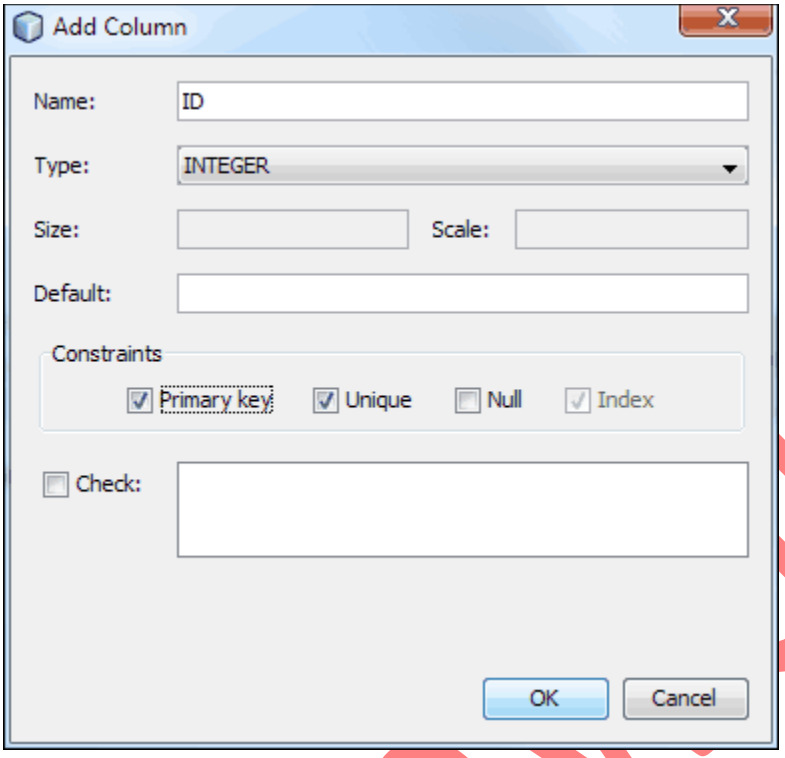
You now need to specify what kind of data is going in to the column. For our ID column, we'll have **Integers**. So scroll along until you come to Data Type. Click on Data Type and a drop down list will appear. From the drop down list, select Integers:



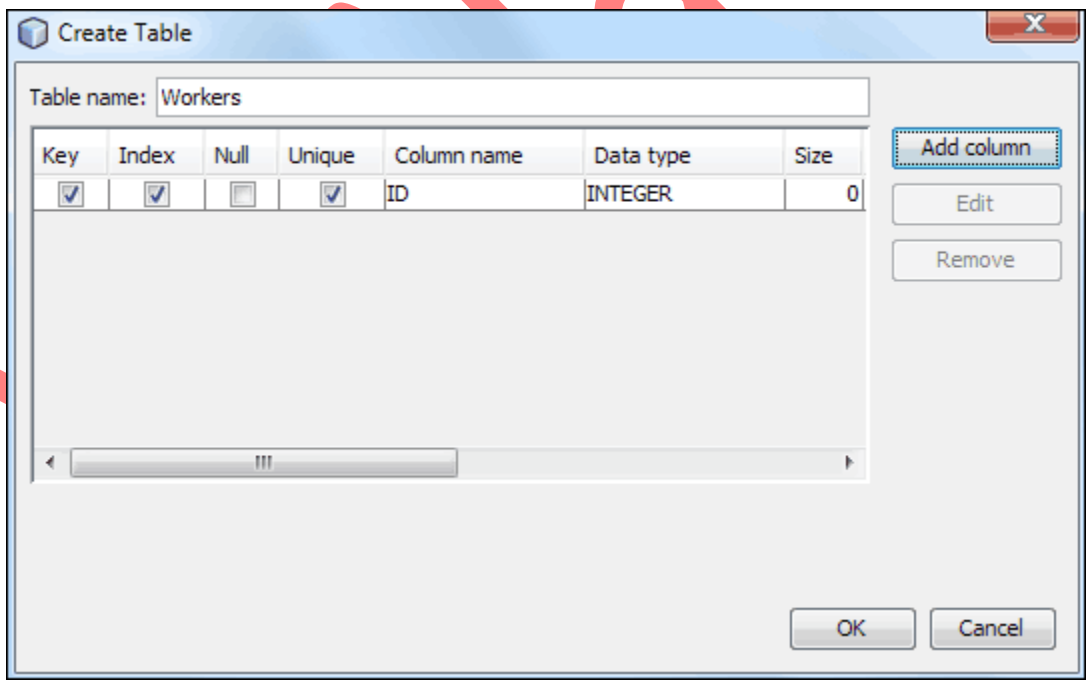
If your dialogue box is like the second one, then you need to click the Add Column button to add your first table column. You'll see another dialogue box appear. This one:



The NAME is the name of the column in the table, like ID, First_Name, etc. The TYPE is the DATA TYPE, Integer, VARCHAR, etc. Click the dropdown list to see more. Then check or uncheck the CONSTRAINTS boxes as indicated below:



Click OK and you should be returned to the Create Table dialogue box:



We now have enough for the ID column in the table. Click the Add Column button on the right to add a new column to the table. Enter the following values for this column (VARCHAR means a variable number of characters):

Key: Unchecked
Index: Unchecked
Null: Unchecked
Unique: Unchecked
Column Name: First_Name
Data Type: VARCHAR
Size: 20

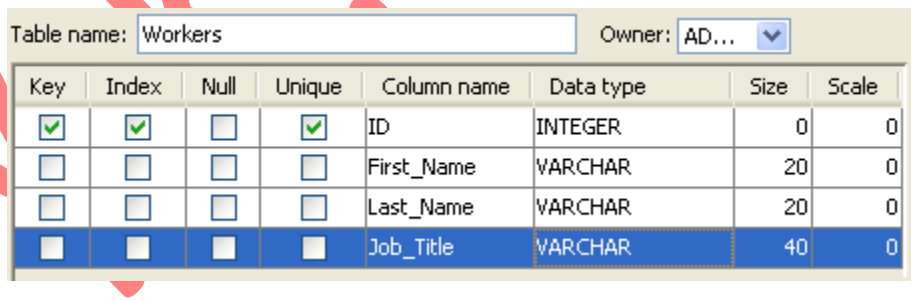
For the third column in your table, enter the following values:

Key: Unchecked
Index: Unchecked
Null: Unchecked
Unique: Unchecked
Column Name: Last_Name
Data Type: VARCHAR
Size: 20

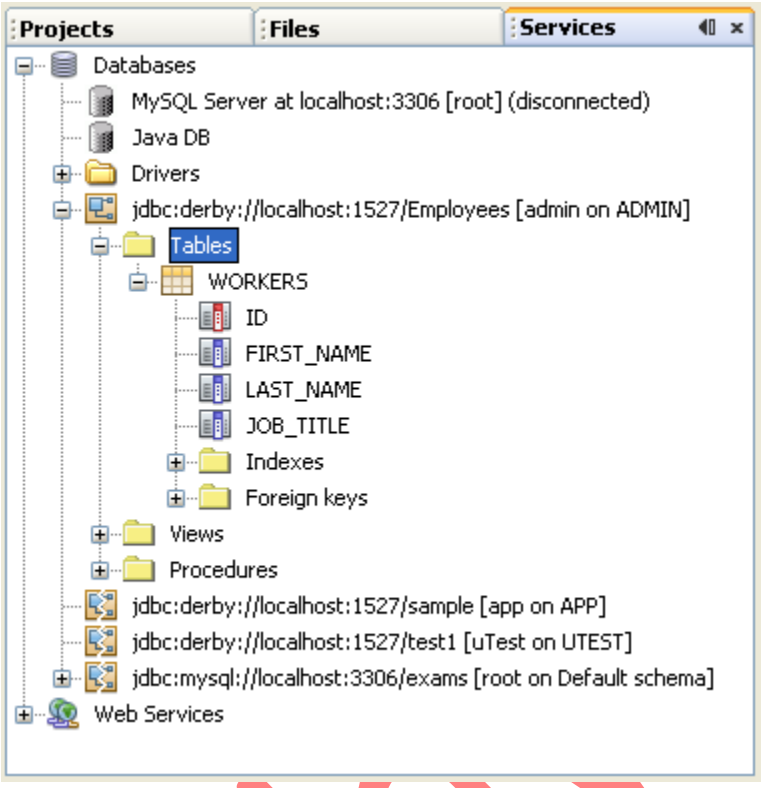
For the final column, here are the values to enter:

Key: Unchecked
Index: Unchecked
Null: Unchecked
Unique: Unchecked
Column Name: Job_Title
Data Type: VARCHAR
Size: 40

When you're finished, your Table dialogue box should look like this:



Click OK when you've entered all the information. Your table and table columns will then be created:

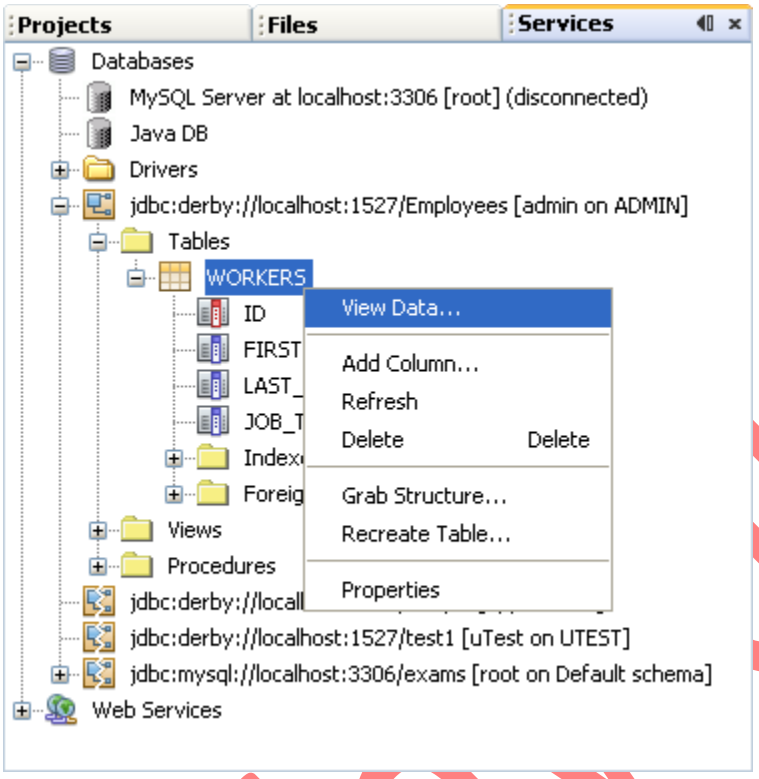


The next thing to do is to add some records to the database table. We'll do that next.

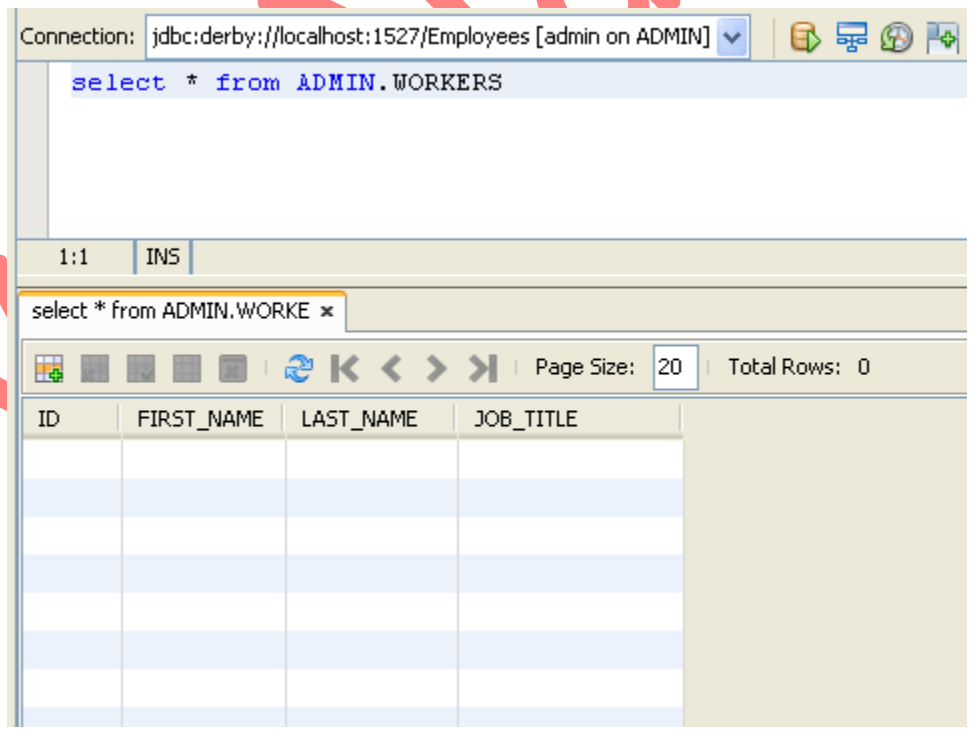
Adding Records to a Java Database Table

A database table is like a spreadsheet, in that it has rows and columns. Each row in our table has cells (fields) for an ID value, a First Name, a Last Name, and a Job Title. Shortly, you'll learn how to write code to add new rows of information to the table. But you can use the NetBeans IDE to add rows as well.

To add a new row to your table, right click on your table name. From the menu that appears, select **View Data**:

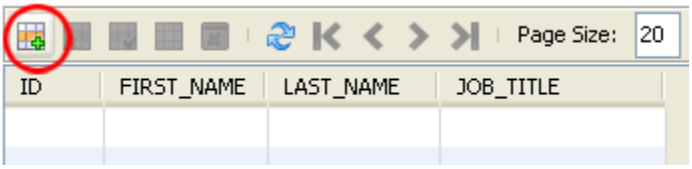


When you click on View Data, you'll see a new window appear in the main NetBeans window

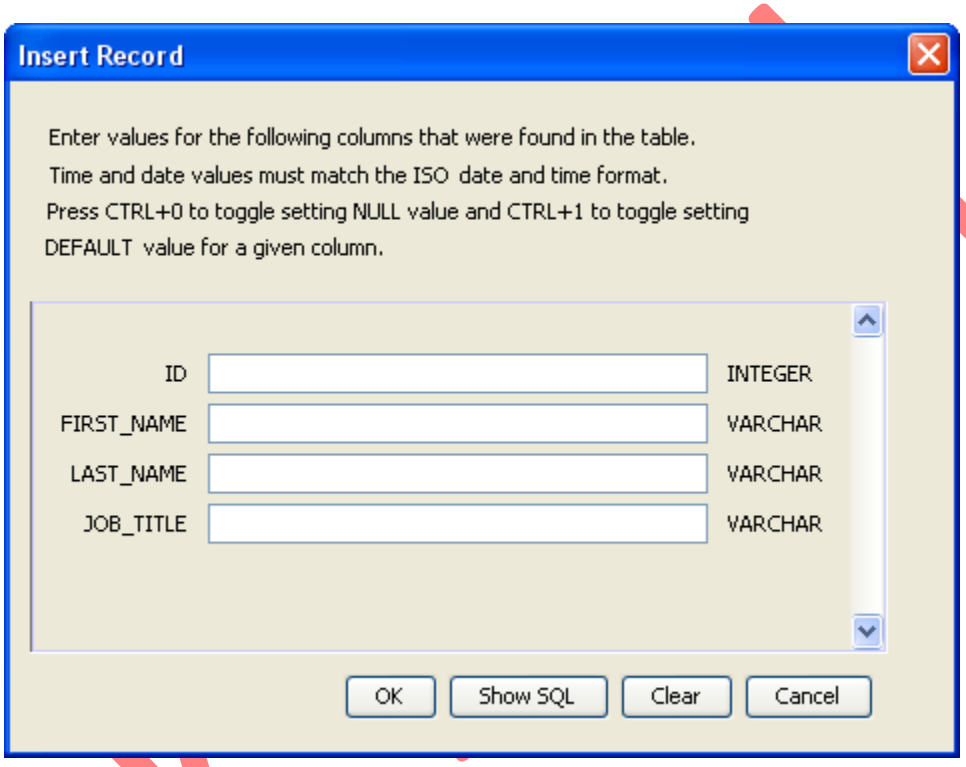


You use the bottom half of window to enter new table rows. The top half is for SQL Commands. (You'll learn more about them soon, when we've finished adding rows.)

To add a new row, click the icon with the green plus symbol, in the bottom half of the window:



When your click the new row icon, a dialogue box appears:

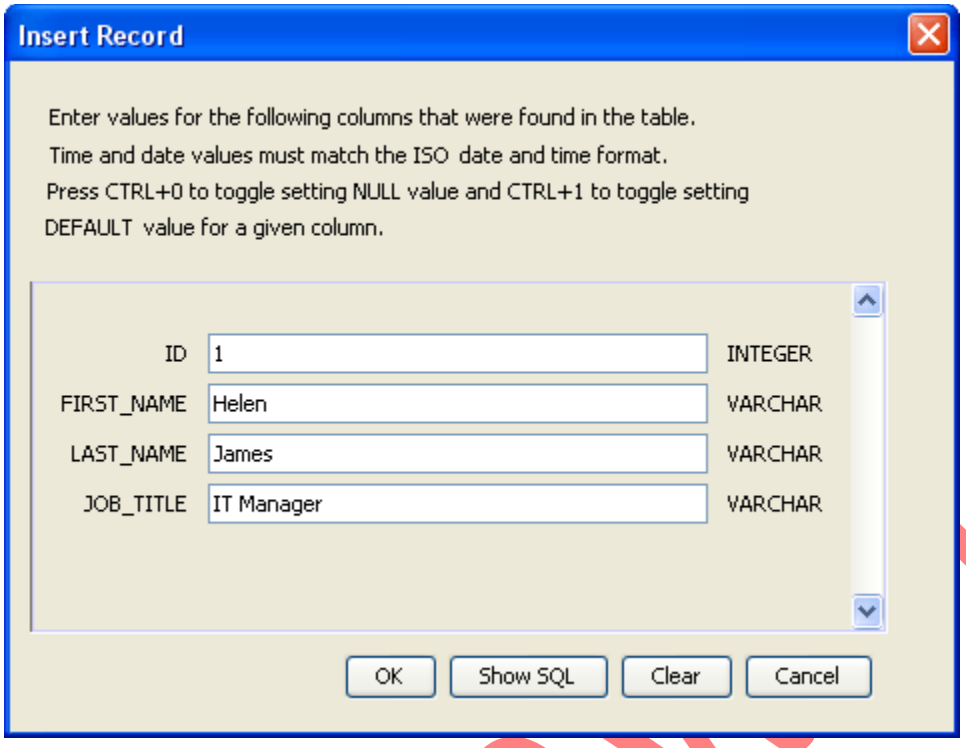


As you can see, there are text boxes for each column in our table. For the ID column, we'll use sequential numbering, starting with 1. The second row in the table will then have an ID of 2, the third row 3, etc. The numbers are not the row numbers: they are just unique values for each ID field. We could have easily started with a value of 100 as the first ID number. The second number would then be 101, the third 102, etc.

Enter the following data as the first row of your table:

ID: 1
First Name: Helen
Last Name: James
Job Title: IT Manager

Your dialogue box will then look like this:



Click OK when you're done and you'll be returned to the NetBeans window. The first row should then be displayed:

ID	FIRST_NAME	LAST_NAME	JOB_TITLE
1	Helen	James	IT Manager

Add three more rows with the following data:

ID: 2
First Name: Eric
Last Name: Khan
Job Title: Programmer

ID: 3
First Name: Tommy
Last Name: Lee
Job Title: Systems Analyst

ID: 4
First Name: Priyanka

Last Name: Collins
Job Title: Programmer

When you've finished adding the new rows, your NetBeans window should look like this one:

ID	FIRST_NAME	LAST_NAME	JOB_TITLE
1	Helen	James	IT Manager
2	Eric	Khan	Programmer
3	Tommy	Lee	Systems Analyst
4	Priyanka	Collins	Programmer

In the next lesson, you'll a few SQL commands.

SQL Commands

In the previous lesson, you created records in your Java DB database table. In this lesson, you'll learn a few SQL commands so that you can manipulate records in a table.

SQL stands for **Structured Query Language**, and is a way to query databases. You can select records, insert, delete, and update records, create tables, drop tables, and more besides. It's quite a powerful tool.

SQL uses simple keywords to do the work. If you want to select all the records from a table, the words SELECT and FROM are used, along with the "all records" symbol, which is *****:

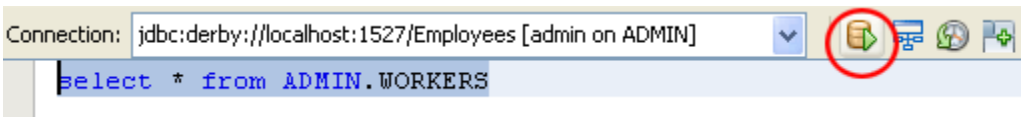
SELECT * FROM table_name

If you look at the top half of the NetBeans window, you'll see that a SELECT statement has already been set up: (NOTE: SQL is not case sensitive)

select * from ADMIN.WORKERS

This says "Select all the records from the table called Workers". (The ADMIN part, before the dot of Workers, is something called a Schema. This describes the structure of the database, but also identifies things like users and the privileges they have. Don't worry about schemas, as we won't be going into them.)

In NetBeans, you run a SQL statement by clicking the **Run** button on the toolbar:



The results from the SQL statements are then displayed in the bottom half of the window:

ID	FIRST_NAME	LAST_NAME	JOB_TITLE
1	Helen	James	IT Manager
2	Eric	Khan	Programmer
3	Tommy	Lee	Systems Analyst
4	Priyanka	Collins	Programmer

The WHERE Clause

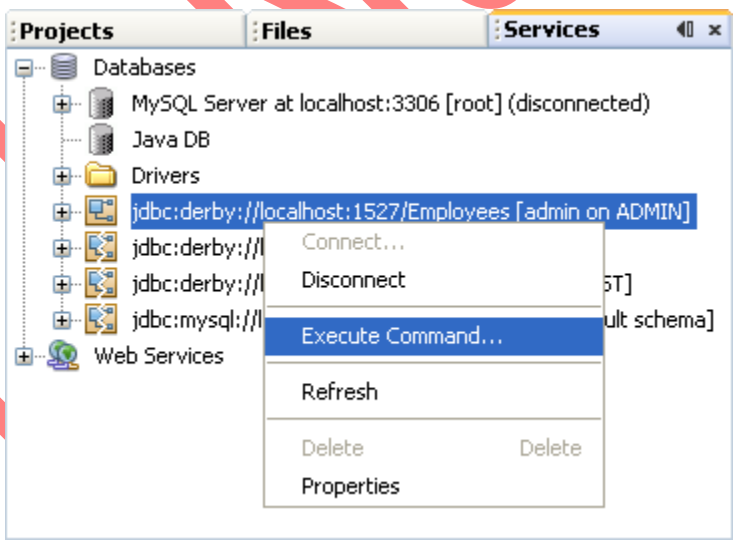
To narrow down your search results, you can use a WHERE clause with the SELECT statement:

SELECT * FROM table_name WHERE column_name=value

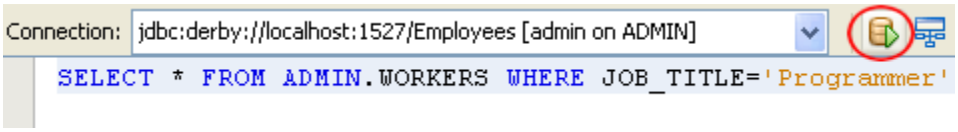
After the keyword WHERE you need the name of a column from your table. You then type an equals sign, followed by a value. As an example, here's a SQL statement that returns all the programmers in our table:

SELECT * FROM ADMIN.WORKERS WHERE JOB_TITLE='Programmer'

To try this SQL Statement out, right-click your table name in the **Services** area. From the menu that appears, select **Execute Command**:



When you click on Execute Command, a new window appears. Type the above SQL Statement, and then click the Run icon:



The results will be displayed in the bottom half of the window:

ID	FIRST_NAME	LAST_NAME	JOB_TITLE
2	Eric	Khan	Programmer
4	Priyanka	Collins	Programmer

As you can see, two rows are returned from the query.

You can also use the keyword LIKE with the WHERE clause. This then replaces the equals sign. LIKE is usually used with a wildcard character. The wildcard character % means "any characters", for example, while an underscore is used for just a single character.

Instead of the equals sign or the keyword LIKE, you can also use the conditional operators (Greater Than, Less Than, etc.). If we had a salary column, we could search for all workers who are getting paid more than 1000 a week:

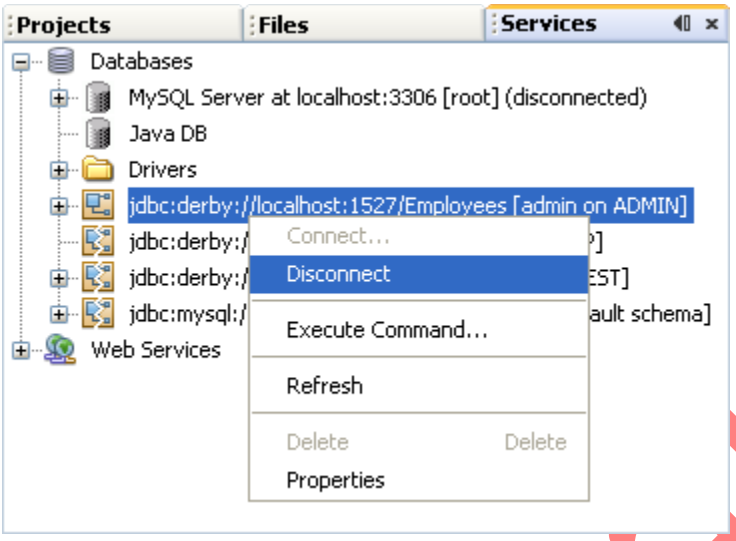
SELECT * FROM ADMIN.WORKERS WHERE SALARY > 1000

Exercise
Try running the following SQL Statement:

SELECT * FROM ADMIN.WORKERS WHERE JOB_TITLE LIKE '%er'

How many results are displayed?

We'll leave SQL Statements there, as we now have enough to start programming. Before doing so, however, close down the connection to your table by right clicking it in the Services are. From the menu, select Disconnect:

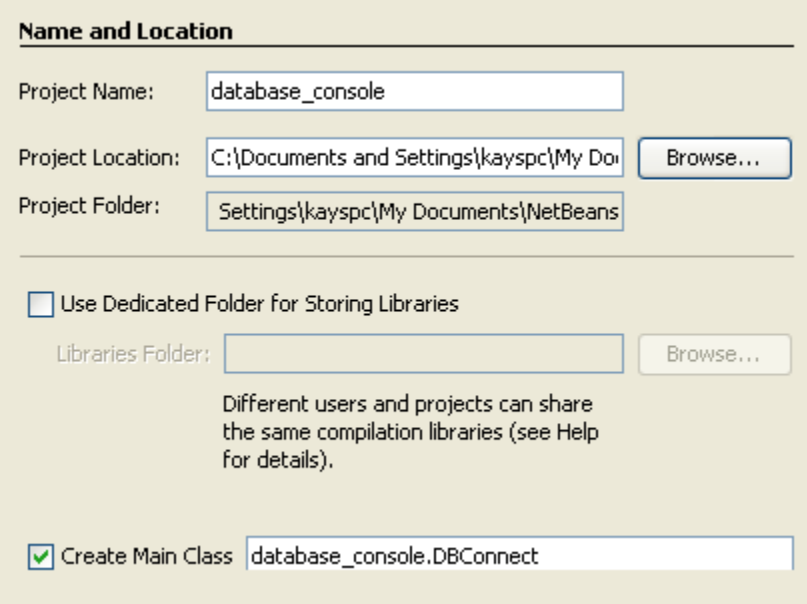


Connect to a Database Using Java Code

In a later section, you'll create a Java form that loads information from a database. The form will have Next and Previous to scroll through the data. Individual records will then be displayed in text fields. We'll also add button to Update a record, Delete a record, and create a new record in the database.

To get started, and for simplicity's sake, we'll use a terminal/console window to output the results from a database.

So start a new project for this by clicking **File > New Project** from the NetBeans menu. Create a **Java Application**. Call the package **database_consol**e, and the Main class **DBConnect**:



When you click Finish, your code should look like this:

```
package database_console;

public class DBConnect {

    public static void main(String[] args) {

    }
}
```

Connecting to the Database

To connect to a database you need a Connection object. The Connection object uses a DriverManager. The DriverManager passes in your database username, your password, and the location of the database.

Add these three import statements to the top of your code:

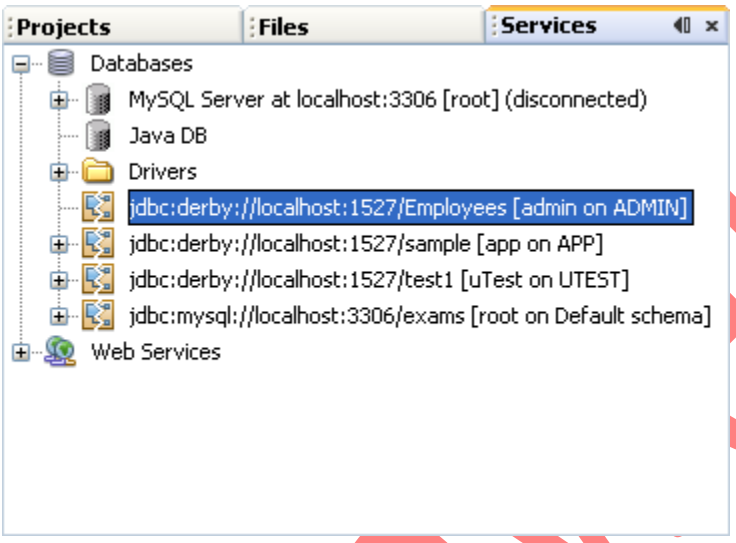
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

To set up a connection to a database, the code is this:

Connection con = DriverManager.getConnection(host, username, password **);**

So the DriverManager has a method called getConnection. This needs a host name (which is the location of your database), a username, and a password. If a connection is successful, a Connection object is created, which we've called **con**.

You can get the host address by looking at the Services tab on the left of NetBeans:



The address of the highlighted database above is:

jdbc:derby://localhost:1527/Employees

The first part, **jdbc:derby://localhost**, is the database type and server that you're using. The 1527 is the port number. The database is **Employees**. This can all go in a String variable:

String host = "jdbc:derby://localhost:1527/Employees";

Two more strings can be added for the username and password:

String uName = "Your_Username_Here";
String uPass= " Your_Password_Here ";

Add these three string before the connection object and your code would look like this:

```
package database_console;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DBConnect {

 public static void main(String[] args) {

  String host = "jdbc:derby://localhost:1527/Employees";
  String uName = "admin";
  String uPass = "admin";
  Connection con = DriverManager.getConnection(host, uName, uPass);

 }
}
```

As you can see in the image above, there is a wavy underline for the Connection code. The reason for this is because we haven't trapped a specific error that will be thrown up when connecting to a database - the SQLException error.

It's the DriverManager that attempts to connect to the database. If it fails (incorrect host address, for example) then it will hand you back a SQLException error. You need to write code to deal with this potential error. In the code below, we're trapping the error in catch part of the **try … catch** statement:

```
try {

}
catch ( SQLException err ) {
System.out.println( err.getMessage( ) );
}
```

In between the round brackets of catch, we've set up a SQLException object called **err**. We can then use the **getMessage** method of this err object.

Add the above **try …catch** block to your own code, and move your four connection lines of code to the try part. Your code will then look like this:

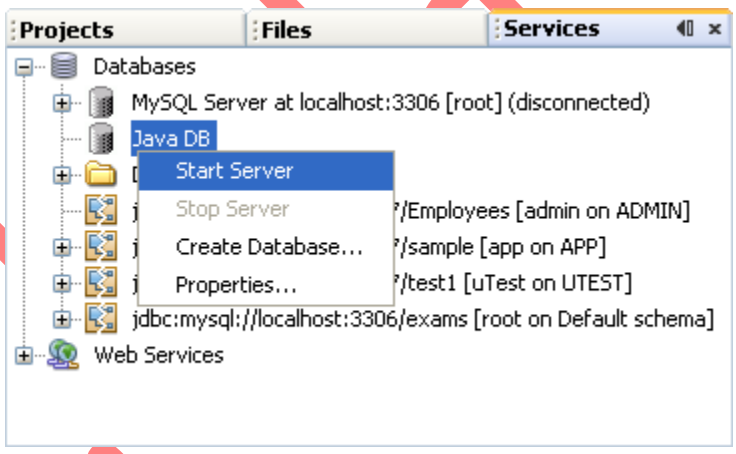
```
public static void main(String[] args) {

  try {
      String host = "jdbc:derby://localhost:1527/Employees";
      String uName = "admin";
      String uPass = "admin";
      Connection con = DriverManager.getConnection(host, uName, uPass);
  }
  catch ( SQLException err ) {
      System.out.println( err.getMessage( ) );
  }

}
```

Try running your code and see what happens.

You may get this error message in the console window:

"java.net.ConnectException : Error connecting to server localhost on port 1527 with message Connection refused: connect."

If you do, it means you haven't connected to your database server. In which case, right click on Java DB in the Services window. From the menu that appears, click Start Server:



You need to make sure that any firewall you may have is not blocking the connection to the server. A good firewall will immediately display a message alerting you that something is trying to get through, and asking if you want to allow or deny it. When you allow the connection, your NetBeans output window should print the following message:

"Apache Derby Network Server - 10.4.1.3 - (648739) started and ready to accept connections on port 1527 at DATE_AND_TIME_HERE"

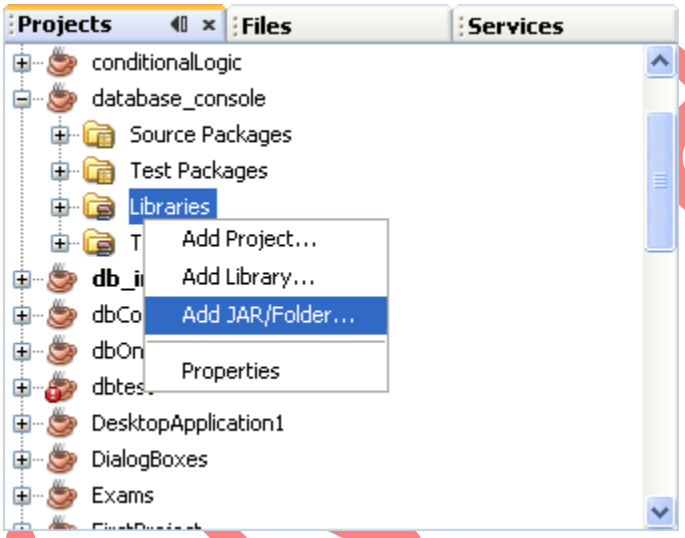
Once your server is started, run the programme again. There's a very good chance you'll get another error message:

"No suitable driver found for jdbc:derby://localhost:1527/Employees"

The reason for this error is that the DriverManager needs a Driver in order to connect to the database. Examples of drivers are Client Drivers and Embedded Drivers. You can import one of these so that the DriverManager can do its job.

Click on the **Projects** tab to the left of the **Services** window in NetBeans. (If you can't see a Projects tab, click **Window > Projects** from the menu bar at the top of NetBeans.)

Locate your project and expand the entry. Right-click **Libraries**. From the menu that appears, select **Add Jar/Folder**:

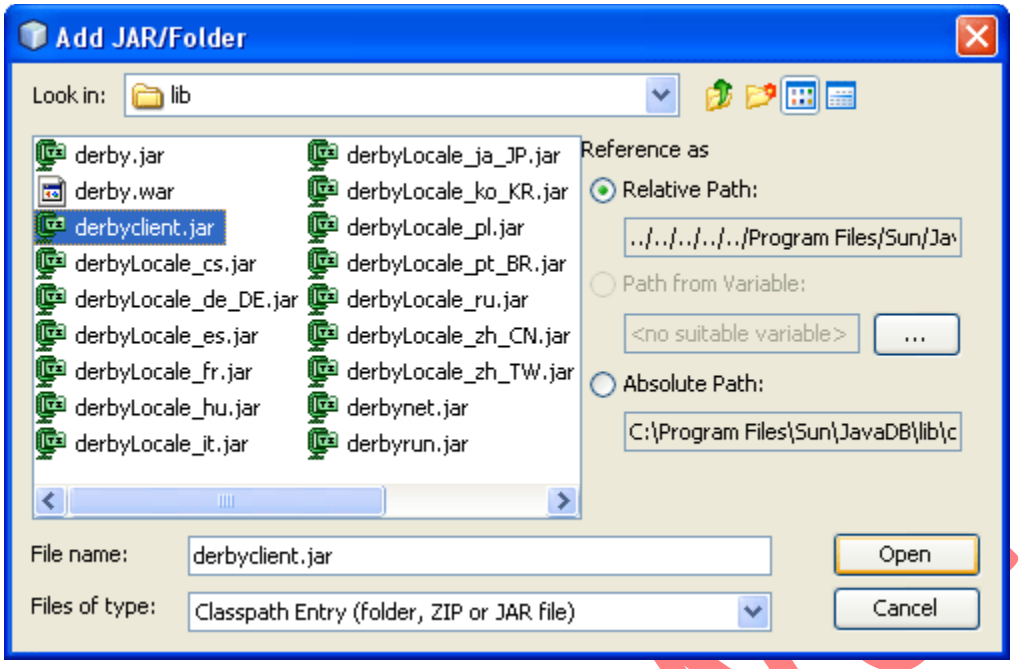


When you click on Add Jar/Folder a dialogue box appears. What you're doing here is adding a Java Archive file to your project. But the JAR file you're adding is for the derby Client Drivers. So you need to locate this folder. On a computer running Windows this will be in the following location:

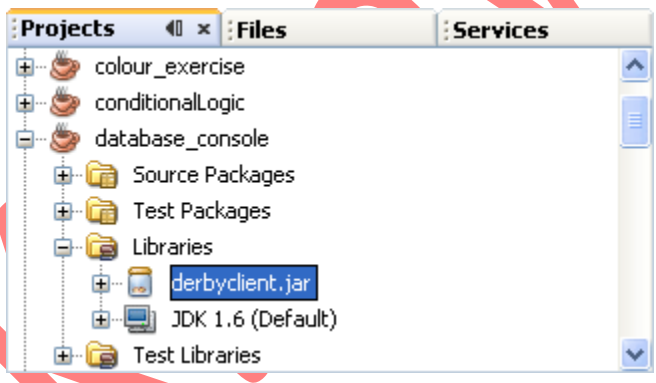
C:\Program Files\Sun\JavaDB\lib

The file you're looking for is called **derbyclient.jar**. If you can't find it, or are using an operating system other than Windows, then do a search for this file. Note the location of the file.

In the dialogue box, select the **derbyclient.jar** file:



Click Open and the file will be added to your project library:



Now that you have a Client driver added to your project, run your programme again. You should now be error free. (The Output window will just say Run, and Build Successful.)

Connecting to a Database Table

Now that you have connected to the database, the next step is to access the table in your database. For this, you need to execute a SQL Statement, and then manipulate all the rows and columns that were returned.

To execute a SQL statement on your table, you set up a Statement object. So add this import line to the top of your code:

import java.sql.Statement;

In the **try** part of the **try … catch** block add the following line (add it just below your Connection line):

Statement stmt = con.createStatement( );

Here, we're creating a Statement object called **stmt**. The Statement object needs a Connection object, with the **createStatment** method.

We also need a SQL Statement for the Statement object to execute. So add this line to your code:

String SQL = "SELECT * FROM Workers";

The above statement selects all the records from the database table called **Workers**.

We can pass this SQL query to a method of the Statement object called **executeQuery**. The Statement object will then go to work gathering all the records that match our query.

However, the **executeQuery** method returns all the records in something called a **ResultSet**. Before we explain what these are, add the following import line to the top of your code:

import java.sql.ResultSet;

Now add this line just below your SQL String line:

ResultSet rs = stmt.executeQuery(SQL);

So our ResultSet object is called **rs**. This will hold all the records from the database table. Before we go any further, though, here's an explanation of what ResultSets are.

ResultSets in Java

A ResultSet is a way to store and manipulate the records returned from a SQL query. ResultSets come in three different types. The type you use depends on what you want to do with the data:

1. Do you just want to move forward through the records, from beginning to end?
2. Do you want to move forward AND backward through the records, as well as detecting any changes made to the records?
   3) Do you want to move forward AND backward through the records, but are not bothered about any changes made to the records?
3. Do you want to move forward AND backward through the records, but are not bothered about any changes made to the records?

Type number 1 on the list above is called a TYPE_FORWARD_ONLY ResultSet. Number 2 on the list is a TYPE_SCROLL_SENSITIVE ResultSet. The third ResultSet option is called TYPE_SCROLL_INSENSITIVE.

The ResultSet type goes between the round brackets of createStement:

Statement stmt = con.createStatement( );

Because we've left the round brackets empty, we'll get the default RecordSet, which is TYPE_FORWARD_ONLY. In the next section, we'll use one of the other types. But you use them like this:

```
Statement stmt = con.createStatement( RecordSet.TYPE_SCROLL_SENSITIVE );
```

So you first type the word RecordSet. After a dot, you add the RecordSet type you want to use.

However, it doesn't end there. If you want to use TYPE_SCROLL_SENSITIVE or TYPE_SCROLL_INSENSITIVE you also need to specify whether the ResultSet is Read Only or whether it is Updatable. You do this with two built-in constants: CONCUR_READ_ONLY and CONCUR_UPDATABLE. Again, these come after the word RecordSet:

ResultSet.**CONCUR_READ_ONLY**
ResultSet.**CONCUR_UPDATABLE**

This leads to a rather long line of code:

```
Statement stmt = con.createStatement( RecordSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
```

One more thing to get used to with ResultSets is something called a **Cursor**. A Cursor is really just a pointer to a table row. When you first load the records into a ResultSet, the Cursor is pointing to just before the first row in the table. You then use methods to manipulate the Cursor. But the idea is to identify a particular row in your table.

Using a ResultSet

Once you have all the records in a Results set, there are methods you can use to manipulate your records. Here are the methods you'll use most often:

next	Moves the Cursor to the next row in your table. If there are no more rows in the table, a value of False will be returned.
previous	Moves the Cursor back one row in your table. If there are no more rows in the table, a value of False will be returned.
first	Moves the Cursor to the first row in your table
last	Moves the Cursor to the last row in your table
absolute	Moves the Cursor to a particular row in the table. So absolute(5) will move the Cursor to row number 5 in the table

The ResultSet also has methods you can use to identify a particular column (field) in a row. You can do so either by using the name of the column, or by using its index number. For our Workers table we set up four columns. They had the following names: ID, First_Name, Last_Name, and Job_Title. The index numbers are therefore 1, 2, 3, 4.

We set up the ID column to hold Integer values. The method you use to get at integer values in a column is getInt:

int id_col = rs.getInt("ID");

Here, we've set up an integer variable called **id_col**. We then use the **getInt** method of our ResultSet object, which is called **rs**. In between the round brackets, we have the name of the column. We could use the Index number instead:

int id_col = rs.getInt(1);

Notice that the Index number doesn't have quote marks, but the name does.

For the other three columns in our database table, we set them up to hold Strings. We, therefore, need the getString method:

String first_name = rs.getString("First_Name");

Or we could use the Index number:

String first_name = rs.getString(2);

Because the ResultSet Cursor is pointing to just before the first record when the data is loaded, we need to use the next method to move to the first row. The following code will get the first record from the table:

```
rs.next( );
int id_col = rs.getInt("ID");
String first_name = rs.getString("First_Name");
String last_name = rs.getString("Last_Name");
String job = rs.getString("Job_Title");
```

Notice that **rs.next** comes first in this code. This will move the Cursor to the first record in the table.

You can add a print line to your code to display the record in the Output window:

```
System.out.println( id_col + " " + first_name + " " + last_name + " " + job );
```

Here's what your code should look like now (we've adapted the print line because it's a bit too long):

```
try {
    String host = "jdbc:derby://localhost:1527/Employees";
    String uName = "admin";
    String uPass = "admin";
    Connection con = DriverManager.getConnection(host, uName, uPass);

    Statement stmt = con.createStatement();
    String sql = "SELECT * FROM Workers";
    ResultSet rs = stmt.executeQuery(sql);

    rs.next();
    int id_col = rs.getInt("ID");
    String first_name = rs.getString("First_Name");
    String last_name = rs.getString("Last_Name");
    String job = rs.getString("Job_Title");

    String p = id_col + " " + first_name + " " + last_name + " " + job;
    System.out.println(p);
}
catch ( SQLException err ) {
    System.out.println( err.getMessage( ) );
}
```

If you want to go through all the records in the table, you can use a loop. Because the next method returns true or false, you can use it as the condition for a while loop:

```
while ( rs.next( ) ) {
```

}

In between the round brackets of **while** we have **rs.next**. This will be true as long as the Cursor hasn't gone past the last record in the table. If it has, rs.next will return a value of false, and the while loop will end. Using rs.next like this will also move the Cursor along one record at a time. Here's the same code as above, but using a while loop instead. Change your code to match:

```
try {
 String host = "jdbc:derby://localhost:1527/Employees";
 String uName = "admin";
 String uPass = "admin";
 Connection con = DriverManager.getConnection(host, uName, uPass);

 Statement stmt = con.createStatement();
 String sql = "SELECT * FROM Workers";
 ResultSet rs = stmt.executeQuery(sql);

 while (rs.next()) {
     int id_col = rs.getInt("ID");
     String first_name = rs.getString("First_Name");
     String last_name = rs.getString("Last_Name");
     String job = rs.getString("Job_Title");

     String p = id_col + " " + first_name + " " + last_name + " " + job;
     System.out.println(p);
 }

 }
 catch ( SQLException err ) {
     System.out.println( err.getMessage( ) );
 }
}
```

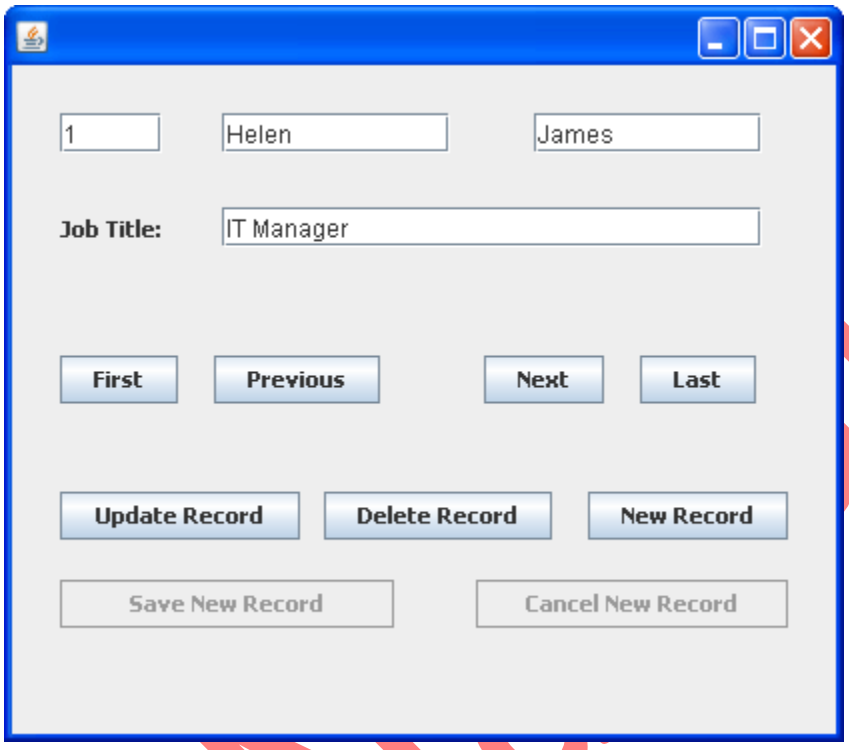
When you run the above code, the Output window should display the following:

```
run:
1 Helen James IT Manager
2 Eric Khan Programmer
3 Tommy Lee Systems Analyst
4 Priyanka Collins Programmer
BUILD SUCCESSFUL (total time: 2 seconds)
```

Now that you have an idea of how to connect to a database table and display records we'll move on and write a more complex programme using forms and buttons to scroll through the records.

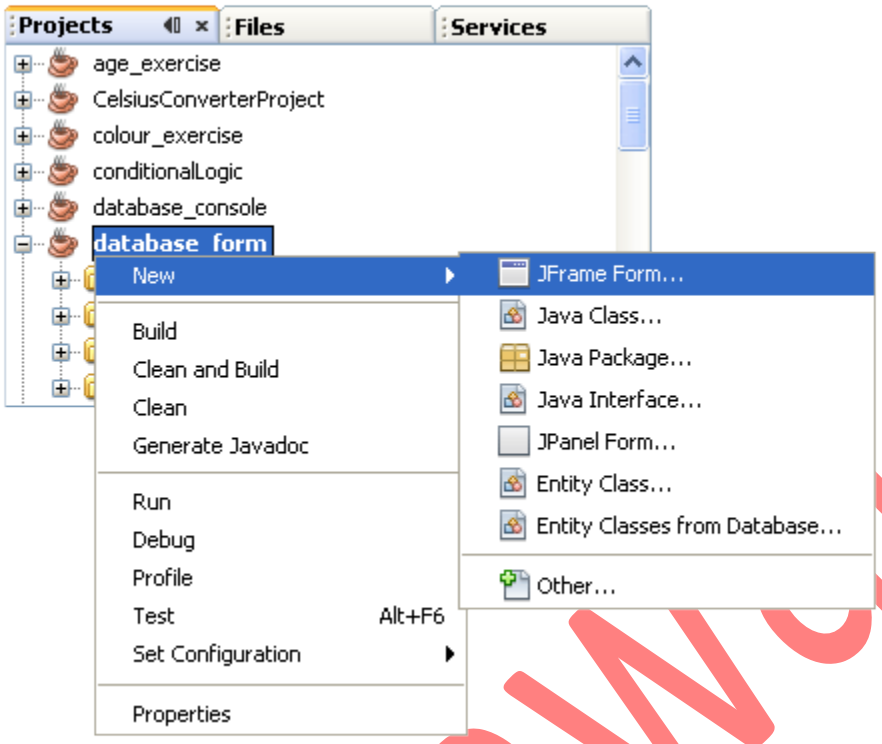
Databases and Java Forms

In this section, you'll create a form with buttons and text fields. The buttons will be used to scroll forwards and backwards through the records in a database table. We'll also add buttons to perform other common database tasks. The form you'll design will look something like this:



Start a new project for this by clicking **File > New Project** from the NetBeans menu. When the dialogue box appears, select **Java > Java Application**. On step one of the dialogue box, type **database_form** as the Project Name. Uncheck the box at the bottom for **Create Main Class**. Click the Finish button to create an empty project.

In the Project area on the left locate your **database_form** project, and right click the entry. From the menu that appears select **New > JFrame Form**:

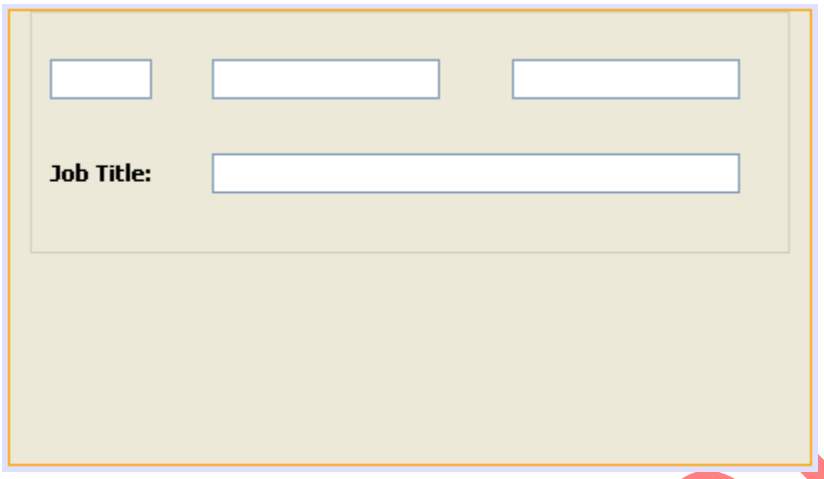


When the dialogue box appears, type **Workers** for the Class name, and **Employees** as the package name. When you click Finish, you should see a blank form appear in the main NetBeans window.

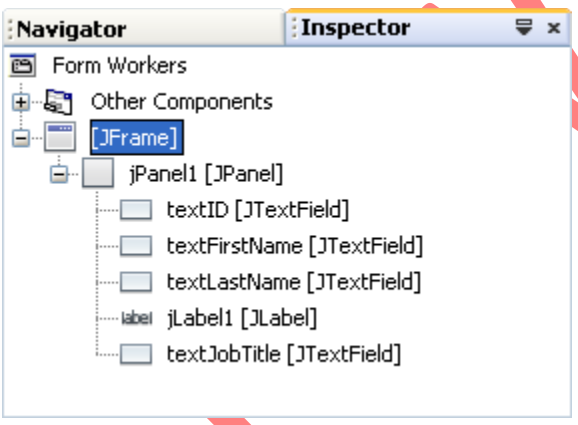
Add a Panel to your form. Then place four Text Fields on the panel. Delete the default text for the Text Fields, leaving them all blank. Change the default variable names for the Text Fields to the following:

textID
textFirstName
textLastName
textJobTitle

Add a label to your panel. Position it just to the left of the job title Text Field. Enter "Job Title" as the text for the label. Arrange the Text Fields and the Label so that your form looks something like this:



Now have a look at the Inspector area to the left of NetBeans. (If you can't see it, click **Window > Inspector** from the NetBeans menu.) It should match ours:



What we want to do now is to have the first record from the database table appear in the text fields when the form first loads. To do that, we can call a method from the form's Constructor.

First, though, we can add Client Driver JAR file to the project, just like last time. This will prevent any "Driver Not Found" errors. So, in the Projects area, right click the Libraries entry for your project. From the menu that appears, select Add JAR/Folder. When the dialogue box appears, locate the derbyclient.jar file. Then click Open to add it to your project.

In the main NetBeans window, click the Source button at the top to get to your code. Now add the following import statements near the top:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;
import javax.swing.JOptionPane;
```

You've met all these before, the first five from the previous section. The last one, JOptionPane, is so that we can display error messages.

Inside of the Class, add the following variable declarations:

Connection con;
Statement stmt;
ResultSet rs;

Just below the Workers Constructor, add the following method:

public void DoConnect( ) {

}

Now add a call to this method from the Constructor:

public Workers() {
initComponents();
DoConnect();
}

Your code window will then look like this: (Don't worry if you have underlines for the import statements. Unless they're red underlines. In which case, you may have made a typing error.)

```
package Employees;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.ResultSet;
import javax.swing.JOptionPane;

public class Workers extends javax.swing.JFrame {

    Connection con;
    Statement stmt;
    ResultSet rs;

    public Workers() {
        initComponents();
        DoConnect();
    }

    public void DoConnect() {

    }
```

What we've done here is to set up a Connection object called **con**, a Statement object called **stmt**, and a ResultSet object called **rs**. We've set them up at the top because our buttons will need access to these objects.

When the form loads, our **DoConnect** method will be called. We can add code here to connect to the database, and display the first record in the text fields.

The code to add for the DoConnect method is almost identical to the code you wrote in the previous section. It's this:

```
public void DoConnect() {
   try {
       //CONNECT TO THE DATABASE
        String host = "jdbc:derby://localhost:1527/Employees";
        String uName = "admin";
        String uPass = "admin";
        con = DriverManager.getConnection(host, uName, uPass);

        //EXECUTE SOME SQL AND LOAD THE RECORDS INTO THE RESULTSET
        stmt = con.createStatement();
        String sql = "SELECT * FROM Workers";
        rs = stmt.executeQuery(sql);

        //MOVE THE CURSOR THE FIRST RECORD AND GET THE DATA
        rs.next();
        int id_col = rs.getInt("ID");
        String id = Integer.toString(id_col);
        String first = rs.getString("First_Name");
        String last = rs.getString("Last_Name");
        String job = rs.getString("Job_Title");

        //DISPLAY THE FIRST RECORD IN THE TEXT FIELDS
        textID.setText(id);
        textFirstName.setText(first);
        textLastName.setText(last);
        textJobTitle.setText(job);
    }
    catch ( SQLException err ) {
        JOptionPane.showMessageDialog(Workers.this, err.getMessage());
    }
}
```

One line that you may not have met is this one:

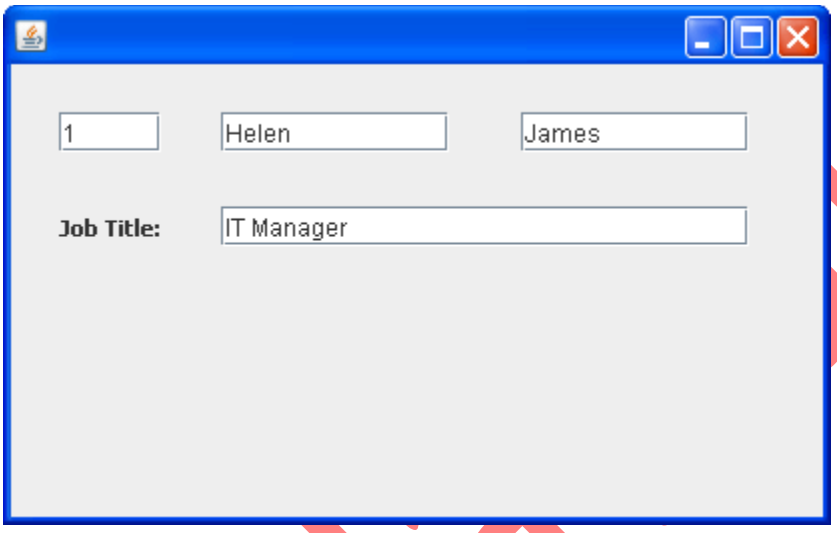
String id = Integer.toString(id_col);

Because the ID column is an Integer, we need to convert it to a String for the **setText** method of the Text Field. We need to do this because Text Field's don't accept Integer values directly - you need to convert them to text.

All the other lines in the code should be familiar to you by now. Study the code to make sure you know what's happening. Then add it your own DoConnect method.

You can run your programme now. First, though, make sure to start your Java DB server from the Services window.

When you run your programme, you should see the first record displayed in the Text Fields:



Now that we have the first record displayed, we can add some buttons to scroll through the rest of the table data. We'll do that in the next lesson.

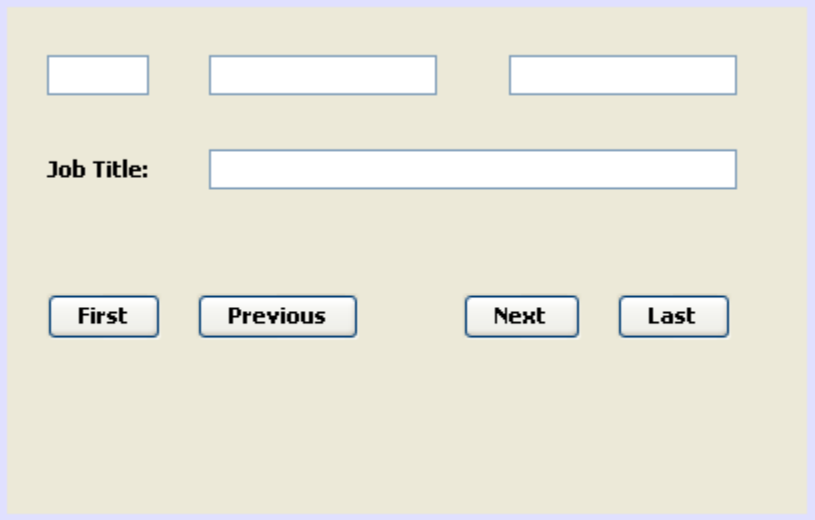
Database Scrolling Buttons

What we'll do now is to add four buttons to the form. The buttons will enable us to move forward through the records, move back, move to the last record, and move to the first record.

So add a new panel to your form. Enlarge it and then add for buttons to the panel. Change the variable names of the buttons to the following:

btnNext
btnPrevious
btnLast
btnFirst

Change the text on each button the Next, Previous, Last, First. You form will then look something like this:



Move to the Next Record

Double click your Next button to create a code stub.

You need to do two things with the Next button: first, check if there is a next record to move to; and second, if there is a next record, display it in the Text Fields. We can create an IF Statement for this. But it needs to be wrapped up in a try … catch block. So add the following to your Next button code stub:

```
try {
if ( rs.next( ) ) {

}
else {
rs.previous( );
JOptionPane.showMessageDialog(Workers.this, "End of File");
}
}
catch (SQLException err) {
JOptionPane.showMessageDialog(Workers.this, err.getMessage());
}
```

The IF Statement moves the ResultSet on one record at a time. If there isn't a next record then a value of false is returned. The Else part moves the ResultSet back one record. That's because the Cursor will have moved past the last record.

In the curly brackets for the IF Statement we can add the code to display the record in the Text Fields:

```
int id_col = rs.getInt("ID");
String id = Integer.toString(id_col);
String first = rs.getString("First_Name");
String last = rs.getString("Last_Name");
String job = rs.getString("Job_Title");

textID.setText(id);
textFirstName.setText(first);
textLastName.setText(last);
textJobTitle.setText(job);
```

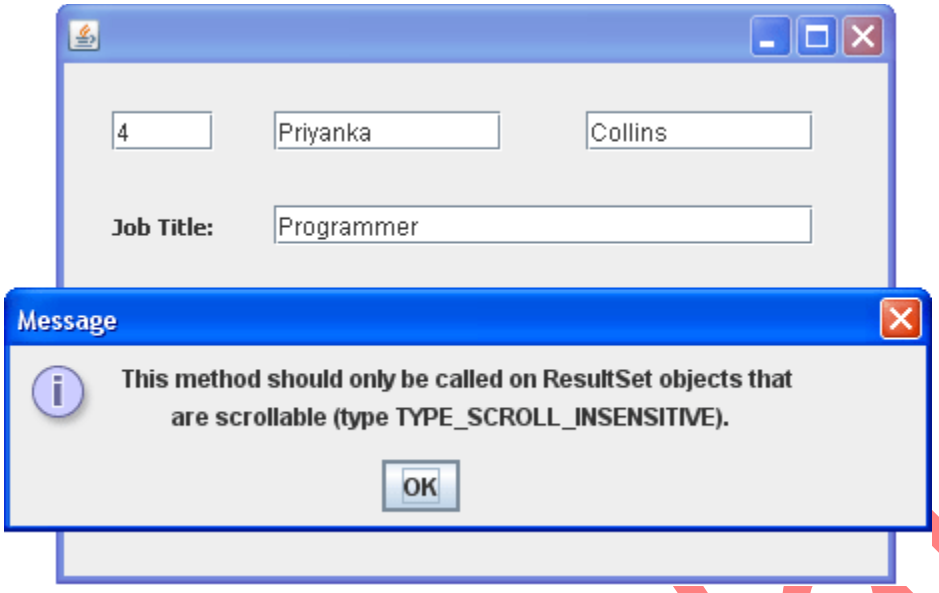
This is the same code we have in our DoConnect method. (We could create a new method, so as not to duplicate any code, but we'll keep it simple.)

The code for your Next button should now look like this:

```
try {
    if (rs.next()) {
        int id_col = rs.getInt("ID");
        String id = Integer.toString(id_col);
        String first = rs.getString("First_Name");
        String last = rs.getString("Last_Name");
        String job = rs.getString("Job_Title");

        textID.setText(id);
        textFirstName.setText(first);
        textLastName.setText(last);
        textJobTitle.setText(job);
    }
    else {
      rs.previous();
      JOptionPane.showMessageDialog(Workers.this, "End of File");
    }
}
catch (SQLException err) {
    JOptionPane.showMessageDialog(Workers.this, err.getMessage());
}
```

When you've added your code, run your programme and test it out. Keep clicking your next button and you'll scroll through all the records in the table. However, there is a problem.

When you get to the last record, you should see an error message appear:



The problem is that we've added an **rs.previous** line. However, we've used the default ResultSet type. As we explained in the last section, this gets you a ResultSet that can only move forward. We can use the type suggested in the error message.

Stop your programme and return to your coding window. In your DoConnect method, locate the following line:

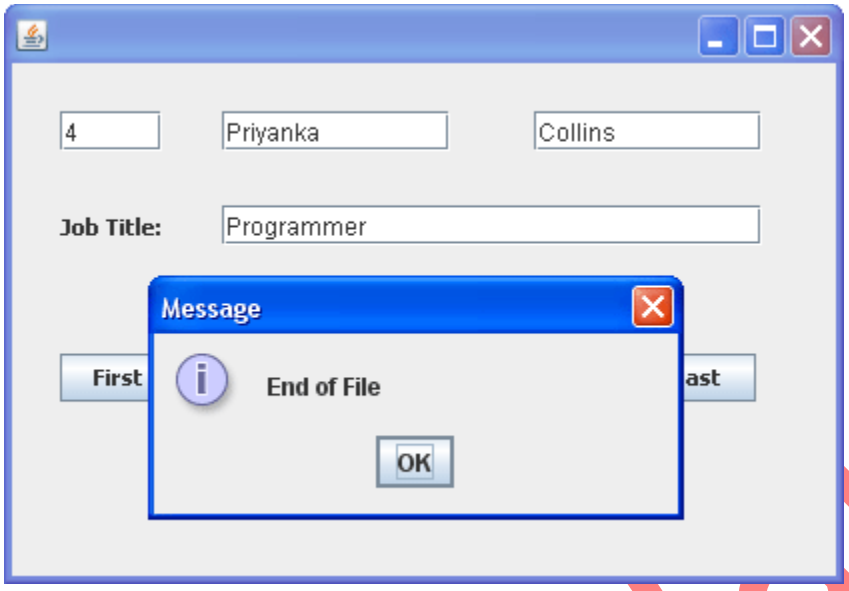
stmt = con.createStatement( );

Change it to this (yours need to go on one line):

stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);

The ResultSet type will now allow us to scroll backwards as well as forwards.

Run your programme again. Click the Next button until you get to the last record. You should see the error message from the **try** part of the **try … catch** block appear:



In the next lesson, you'll learn how to backwards through your database records.

Move Back through a Java Database

The code for the **Previous** button is similar to the **Next** button. But instead of using **rs.Next**, you use **rs.Previous**.

Return to the Design window and double click your **Previous** button to create a code stub.

Instead of typing out all that code again, simply copy and paste the code from your Next button. Then change the **rs.Next**, in the IF statement to **rs.Previous**. Change the rs.Previous in the ELSE part to rs.Next. You can also change your error message text from "End of File" to "Start of File".

Your code should look like this:

```
try {
    if (rs.previous()) {
        int id_col = rs.getInt("ID");
        String id = Integer.toString(id_col);
        String first = rs.getString("First_Name");
        String last = rs.getString("Last_Name");
        String job = rs.getString("Job_Title");

        textID.setText(id);
        textFirstName.setText(first);
        textLastName.setText(last);
        textJobTitle.setText(job);
    }
    else {
      rs.next();
      JOptionPane.showMessageDialog(Workers.this, "Start of File");
    }
}
catch (SQLException err) {
    JOptionPane.showMessageDialog(Workers.this, err.getMessage());
}
```

Run your programme again. You should be able to move backward and forward through the database by clicking your two buttons.

Move to the First and Last Records

Moving to the first and last records of your database is a lot easier.

Double click your **First** button to create the code stub. Now add the following code:

```
try {
    rs.first();
    int id_col = rs.getInt("ID");
    String id = Integer.toString(id_col);
    String first = rs.getString("First_Name");
    String last = rs.getString("Last_Name");
    String job = rs.getString("Job_Title");

    textID.setText(id);
    textFirstName.setText(first);
    textLastName.setText(last);
    textJobTitle.setText(job);
}
catch (SQLException err) {
    JOptionPane.showMessageDialog(Workers.this, err.getMessage());
}
```

We have no need of an IF ... ELSE Statement, now. The only thing we need to do is move the Cursor to the first record with **rs.First**, then display the first record in the Text Fields.

Similarly, add the following code for your Last button (you can copy and paste the code for the First button):

```
try {
    rs.last();
    int id_col = rs.getInt("ID");
    String id = Integer.toString(id_col);
    String first = rs.getString("First_Name");
    String last = rs.getString("Last_Name");
    String job = rs.getString("Job_Title");

    textID.setText(id);
    textFirstName.setText(first);
    textLastName.setText(last);
    textJobTitle.setText(job);
}
catch (SQLException err) {
    JOptionPane.showMessageDialog(Workers.this, err.getMessage());
}
```

The only change to make is the use of rs.Last on the first line in place of rs.First.

When you've added the code, run your programme again. You should now be able to jump to the last record in your database, and jump to the first record.

In the next part, you'll learn how to update a record.

Updating a Record

The ResultSet has **Update** methods that allow you to update records not only in the ResultSet itself, but in the underlying database. Let's see how it works.

Make your form a bit longer. Now add a new panel to the form. Add a new button to the panel. Change the default variable name to **btnUpdateRecord**. Change the text on the button to **Update Record**. We're also going to have buttons to create a new record in the database, to save a record, cancel any updates, and to delete a record. So add four more buttons to the panel. Make the following changes:

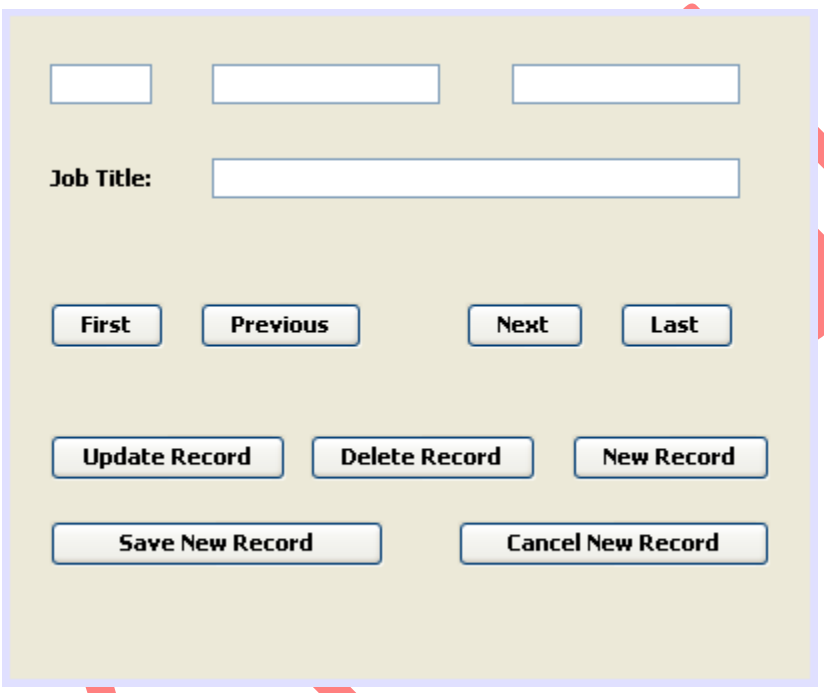
Button Variable Name: btnNewRecord
Button Text: New Record

Button Variable Name: btnDeleteRecord
Button Text: Delete Record

Button Variable Name: btnSaveRecord
Button Text: Save New Record

Button Variable Name: btnCancelNewRecord
Button Text: Cancel New Record

When you're done, your form should look something like this one (though feel free to rearrange the buttons):



Double click your Update button to create a code stub.

The first thing to do is get the text from the Text Fields:

```
String first = textFirstName.getText( );
String last = textLastName.getText( );
String job = textJobTitle.getText( );
String ID = textID.getText( );
```

If we want to update an ID field, however, we need to convert the String to an Integer:

```
int newID = Integer.parseInt( ID );
```

The Integer object has a method called **parseInt**. In between the round brackets of parseInt, you type the string that you're trying to convert.

Now that we have all the data from the Text Fields, we can call the relevant update methods of the ResultSet object:

rs.updateString("First_Name", first);

There are quite a few different update methods to choose from. The one above uses **updateString**. But you need the field type from your database table here. We have three strings (First_Name, Last_Name, Job_Title) and one integer value (ID). So we need three **updateString** methods and one **updateInt**.

In between the round brackets of the update methods, you need the name of a column from your database (though this can be its Index value instead). After a comma you type the replacement data. So, in the example above, we want to update the First_Name column and replace it with the value held in the variable called first.

The update methods just update the ResultSet, however. To commit the changes to the database, you issue an **updateRow** command:

rs.updateRow( );

Here are all the lines of code to update the ResultSet and the database table:

```
try {
rs.updateInt( "ID", newID );
rs.updateString( "First_Name", first );
rs.updateString( "last_Name", last );
rs.updateString( "Job_Title", job );
rs.updateRow( );
JOptionPane.showMessageDialog(Workers.this, "Updated");
}
catch (SQLException err) {
System.out.println(err.getMessage() );
}
```

Again, we need to wrap it all up in a **try … catch** statement, just in case something goes wrong. Notice, too, that we've added a message box for a successful update.

Here's the entire code to add for your Update Button:

```
String first = textFirstName.getText();
String last = textLastName.getText();
String job = textJobTitle.getText();
String ID = textID.getText();

int newID = Integer.parseInt(ID);

try {
    rs.updateInt("ID", newID);
    rs.updateString("First_Name", first);
    rs.updateString("last_Name", last);
    rs.updateString("Job_Title", job);
    rs.updateRow();
    JOptionPane.showMessageDialog(Workers.this, "Updated");
}
catch (SQLException err) {
    System.out.println(err.getMessage());
}
```

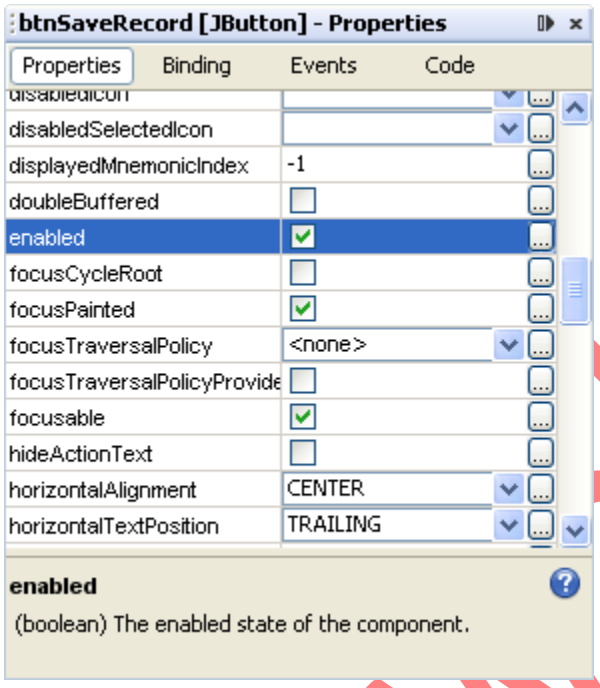
Run your programme and try it out. Change some data in a Text Field (Tommy to Timmy, for example). Then click your Update button. Scroll past the record then go back. The change should still be there. Now close down your programme and run it again. You should find that the changes are permanent.

In the next lesson, you'll see how to add a new record

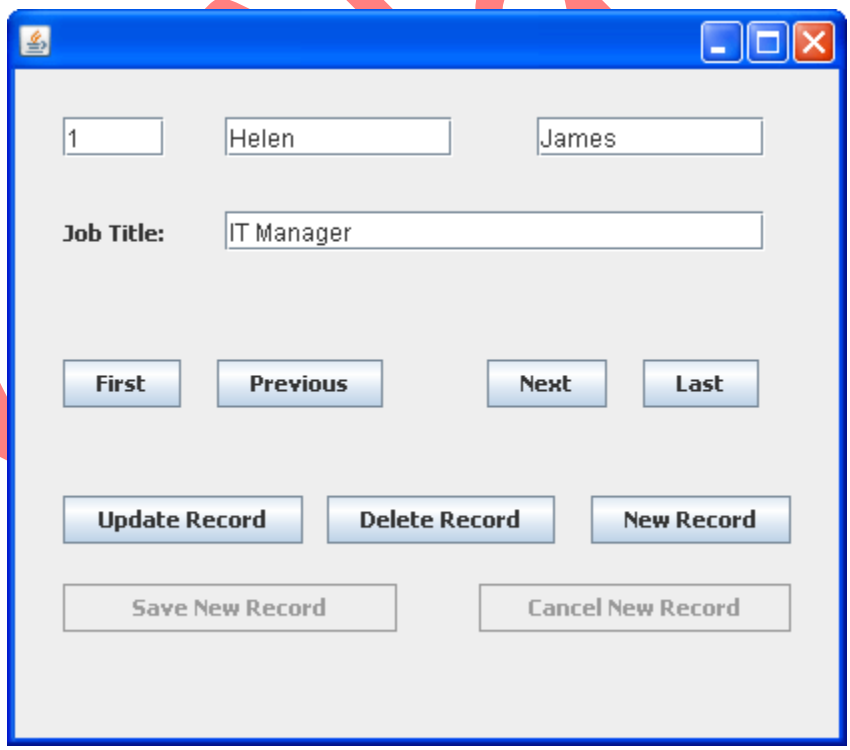
Add a New Record

We have three buttons that refer to new records: New Record, Save New Record, and Cancel New Record. The New Record button will only clear the Text Fields, and ready them for new data to be entered. We can also disable some other buttons, including the New Record button. Another thing we can do is to make a note of which record is currently loaded. If a user changes his or her mind, we can enable all the buttons again by clicking the Cancel button. Clicking the Save New Record button will do the real work of saving the data to the database.

If that's all a little confusing, try the following. Click on your Save New Record button to select it. In the Properties area on the right, locate the Enabled property:



Uncheck the box to the right of enabled. The Save New Record will be disabled. Do the same for the Cancel New Record button. The Cancel New Record will be disabled. When your form loads, it will look like this:



Even if you had code for these two buttons, nothing would happen if you clicked on either of them.

When the New Record button is clicked, we can disable the following buttons:

First
Previous
Next
Last
Update Record
Delete Record
New Record

The Save and Cancel buttons, however, can be enabled. If the user clicks Cancel, we can switch the buttons back on again.

Double click your New Record button to create a code stub. Add the following lines of code:

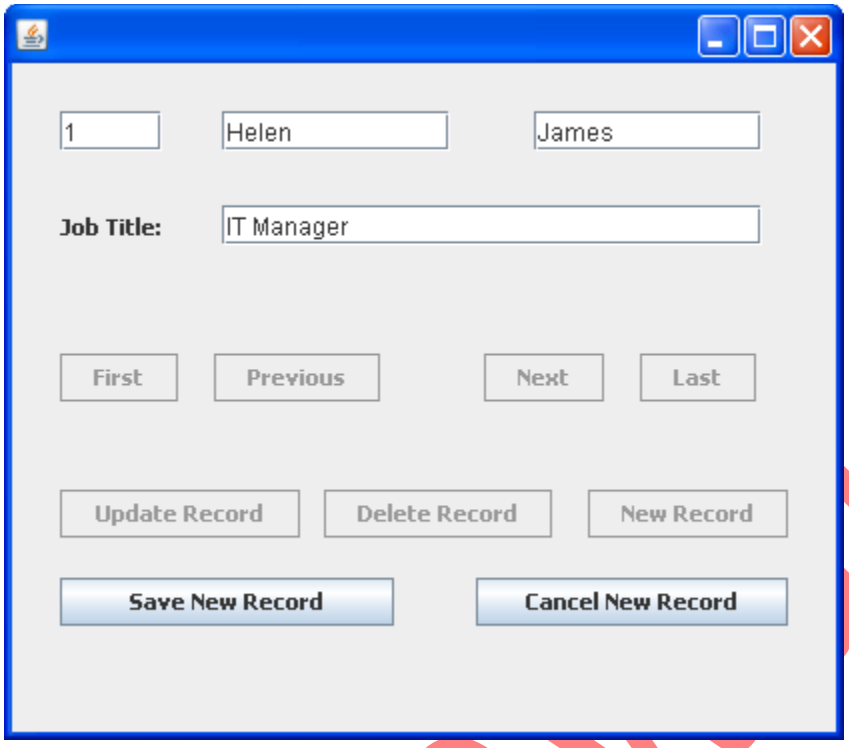
```
btnFirst.setEnabled( false );
btnPrevious.setEnabled( false ) ;
btnNext.setEnabled( false );
btnLast.setEnabled( false );
btnUpdateRecord.setEnabled( false );
btnDelete.setEnabled( false );
btnNewRecord.setEnabled( false );

btnSaveRecord.setEnabled( true );
btnCancelNewRecord.setEnabled( true );
```

So seven of the buttons get turned off using the setEnabled property. Two of the buttons get turned on.

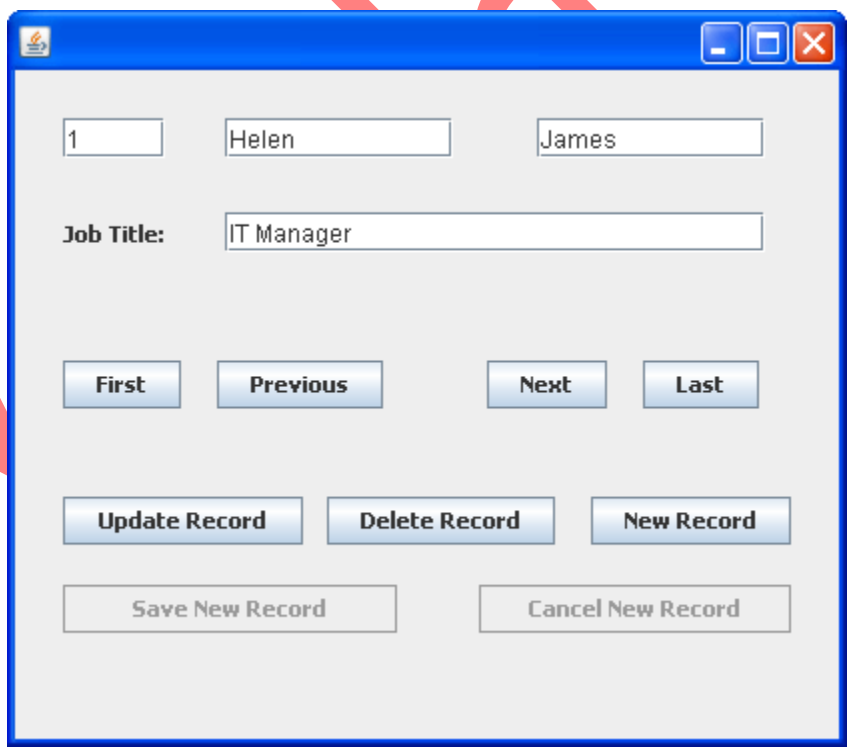
We can do the reverse for the Cancel button. Switch back to Design view. Double click your Cancel New Record button to create a code stub. Add the following:

```
btnFirst.setEnabled( true );
btnPrevious.setEnabled( true ) ;
btnNext.setEnabled( true );
btnLast.setEnabled( true );
btnUpdateRecord.setEnabled( true );
btnDelete.setEnabled( true );
btnNewRecord.setEnabled( true );

btnSaveRecord.setEnabled( false );
btnCancelNewRecord.setEnabled( false );
```

Now run your programme and test it out. Click the New Record button and the form will look like this:



Click the Cancel New Record button and the form will look like this:



Another thing we need to do is to record which row is currently loaded. In other words, which row number is currently loaded in the Text Fields. We need to do this because the Text Fields are going to be cleared. If the Cancel button is clicked, then we can reload the data that was erased.

Add the following Integer variable to the top of your code, just below your Connection, Statement, and ResultSet lines:

int curRow = 0;

The top of your code should look like this:

```
public class Workers extends javax.swing.JFrame {

    Connection con;
    Statement stmt;
    ResultSet rs;
    int curRow = 0;

    public Workers() {
        initComponents();
        DoConnect();
    }
```

Now go back to your New Record code.

To get which row the Cursor is currently pointing to there is a method called getRow. This allows you to store the row number that the Cursor is currently on:

curRow = rs.getRow( );

We'll use this row number in the Cancel New Record code.

The only other thing we need to do for the New Record button is to clear the Text Fields. This is quite simple:

textFirstName.setText("");
textLastName.setText("");
textJobTitle.setText("");
textID.setText("");

So we're just setting the Text property to a blank string.

Because we've used a method of the ResultSet, we need to wrap everything up in a **try … catch** block. Here's what the code should look like for your New Record button:

```
try {
    curRow = rs.getRow();

    textFirstName.setText("");
    textLastName.setText("");
    textJobTitle.setText("");
    textID.setText("");

    btnFirst.setEnabled(false);
    btnPrevious.setEnabled(false);
    btnNext.setEnabled(false);
    btnLast.setEnabled(false);
    btnUpdateRecord.setEnabled(false);
    btnDelete.setEnabled(false);
    btnNewRecord.setEnabled(false);

    btnSaveRecord.setEnabled(true);
    btnCancelNewRecord.setEnabled(true);
}
catch (SQLException err) {
    System.out.println(err.getMessage());
}
```

For the Cancel button, we need to get the row that was previously loaded and put the data back in the Text Fields.

To move the Cursor back to the row it was previously pointing to, we can use the absolute method:

rs.absolute(curRow);

The absolute method moves the Cursor to a fixed position in the ResultSet. We want to move it the value that we stored in the variable curRow.

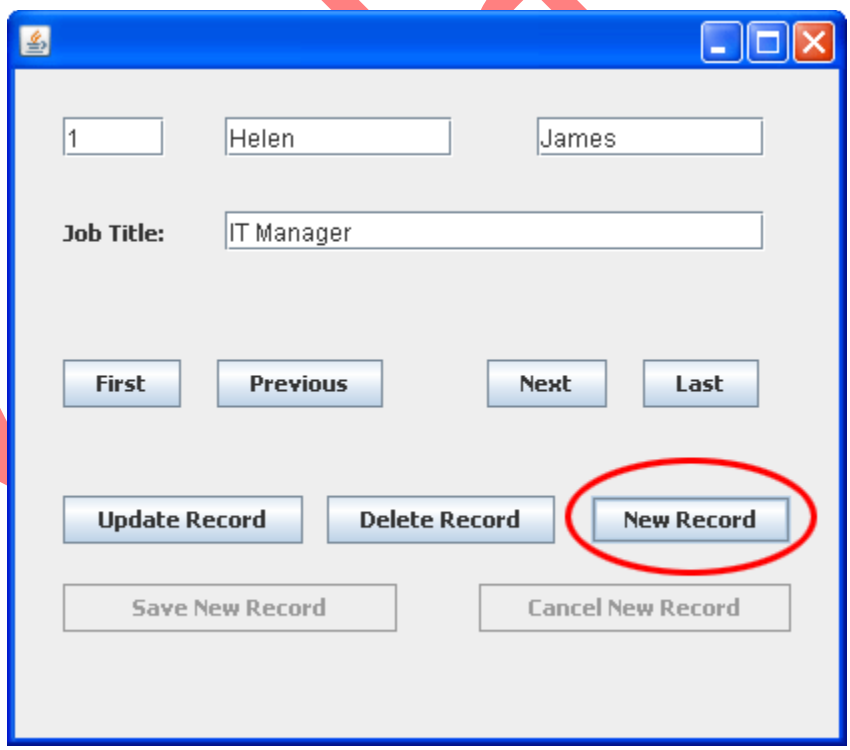
Now that Cursor is pointing at the correct row, we can load the data into the Text Fields:

textFirstName.setText(rs.getString("First_Name"));
textLastName.setText(rs.getString("Last_Name"));
textJobTitle.setText(rs.getString("Job_Title"));
textID.setText(Integer.toString(rs.getInt("ID")));

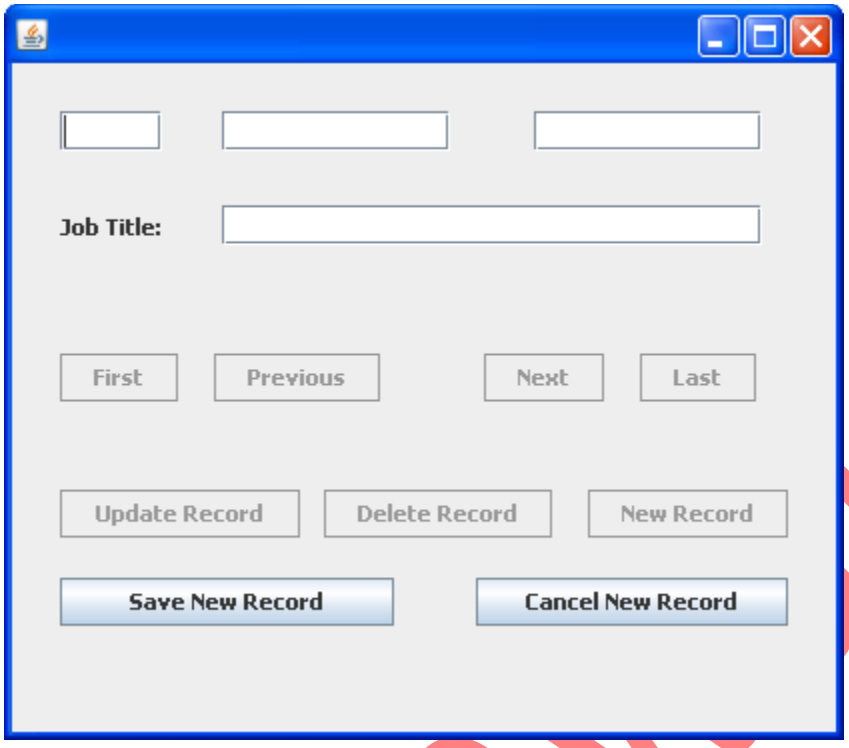
Wrapping it all in a **try … catch** block gives us the following code:

```
try {
    rs.absolute(curRow);
    textFirstName.setText(rs.getString("First_Name"));
    textLastName.setText(rs.getString("Last_Name"));
    textJobTitle.setText(rs.getString("Job_Title"));
    textID.setText( Integer.toString(rs.getInt("ID")) );

    btnFirst.setEnabled(true);
    btnPrevious.setEnabled(true);
    btnNext.setEnabled(true);
    btnLast.setEnabled(true);
    btnUpdateRecord.setEnabled(true);
    btnDelete.setEnabled(true);
    btnNewRecord.setEnabled(true);

    btnSaveRecord.setEnabled(false);
    btnCancelNewRecord.setEnabled(false);
}
catch (SQLException err) {
    System.out.println(err.getMessage());
}
```

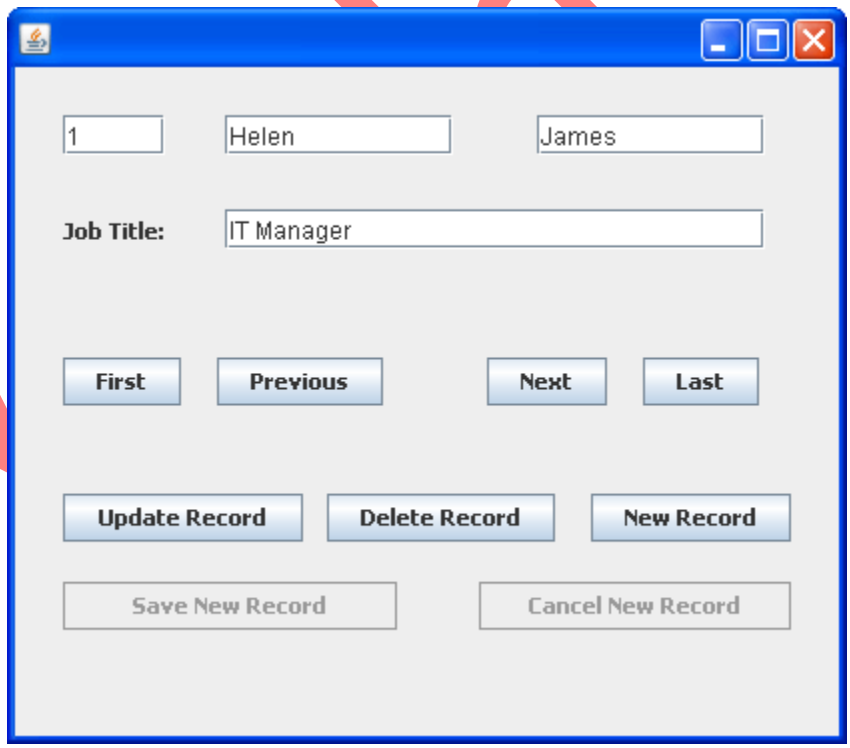
When you've finished adding the code for the New and Cancel buttons, run your programme and try it out. Before clicking the New Record button, the form will look like this:



Click the New Record button to see the Text Fields cleared:



Clicking the Cancel button will reload the data:



Now that the New and Cancel buttons have been set up, we can go ahead and save any new data entered into the Text Fields.

Save a New Record

Before you can save a new record, you have to move the Cursor to something called the **Insert Row**. This creates a blank record in the ResultSet. You then add the data to the ResultSet:

```
rs.moveToInsertRow( );

rs.updateInt("ID", newID);
rs.updateString("First_Name", first);
rs.updateString("Last_Name", last);
rs.updateString("Job_Title", job);

rs.insertRow( );
```

After adding the data to the ResultSet, the final line inserts a new row.

However, to commit any changes to the database what we'll do is to close our Statement object and our ResultSet object. We can then reload everything. If we don't do this, there's a danger that the new record won't get added, either to the ResultSet or the database. (This is due to the type of Driver we've used.)

To close a Statement or a ResultSet, you just issue the close command:

```
stmt.close( );
rs.close( );
```

The code to reload everything is the same as the code you wrote when the form first loads:

```
stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);

String sql = "SELECT * FROM Workers";
rs = stmt.executeQuery(sql);

rs.next( );
int id_col = rs.getInt("ID");
String id = Integer.toString(id_col);
String first2 = rs.getString("First_Name");
String last2 = rs.getString("Last_Name");
String job2 = rs.getString("Job_Title");

textID.setText(id);
textFirstName.setText(first2);
textLastName.setText(last2);
textJobTitle.setText(job2);
```

You're not doing anything different, here: just selecting all the records again and putting the first one in the Text Fields.

Here's all the code that saves a new record to the database (Obviously, a lot of this code could have went into a method of its own):

```
String first = textFirstName.getText();
String last = textLastName.getText();
String job = textJobTitle.getText();
String ID = textID.getText();
int newID = Integer.parseInt(ID);

try {
    rs.moveToInsertRow();

    rs.updateInt("ID", newID);
    rs.updateString("First_Name", first);
    rs.updateString("Last_Name", last);
    rs.updateString("Job_Title", job);

    rs.insertRow();

    stmt.close();
    rs.close();

    stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                               ResultSet.CONCUR_UPDATABLE);
    String sql = "SELECT * FROM Workers";
    rs = stmt.executeQuery(sql);

    rs.next();
    int id_col = rs.getInt("ID");
    String id = Integer.toString(id_col);
    String first2 = rs.getString("First_Name");
    String last2 = rs.getString("Last_Name");
    String job2 = rs.getString("Job_Title");

    textID.setText(id);
    textFirstName.setText(first2);
    textLastName.setText(last2);
    textJobTitle.setText(job2);

    btnFirst.setEnabled(true);
    btnPrevious.setEnabled(true);
    btnNext.setEnabled(true);
    btnLast.setEnabled(true);
    btnUpdateRecord.setEnabled(true);
    btnDelete.setEnabled(true);
    btnNewRecord.setEnabled(true);

    btnSaveRecord.setEnabled(false);
    btnCancelNewRecord.setEnabled(false);

}
catch (SQLException err) {
    System.out.println(err.getMessage());
}
```

The code is a bit long, but you can copy and paste a lot of it from your **DoConnect** method. (We've Photo-shopped the **stmt** line because it's too big to fit on this page. Yours should go on one line).

(One other issue is that the ID column needs to be unique. Ideally, you'd write a routine to get the last ID number, then add one to it. Other databases, like MySql, have an AutoIncrement value to take care of these things. Just make sure that the ID value isn't one you have used before, otherwise you'll get an error message. Or write a routine to get a unique ID!)

Run your programme and test it out. You now be able to save new records to your database.

In the next lesson, you learn about deleting records

Delete a Record

Deleting a row can be straightforward: Just use deleteRow method of the ResultSet:

rs.deleteRow( );

However, the Driver we are using, the ClientDriver, leaves a blank row in place of the data that was deleted. If you try to move to that row using your Next or Previous buttons, the ID Text Field will have a 0 in it, and all the others will be blank.

To solve this problem we'll first delete a row then, again, close the Statement object and the ResultSet objects. We can then reload all the data in the Text Fields. That way, we won't have any blank rows.

Here's the code to add for your Delete Record button:

```
try {
    rs.deleteRow();
    stmt.close();
    rs.close();
    stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                               ResultSet.CONCUR_UPDATABLE);
    String sql = "SELECT * FROM Workers";
    rs = stmt.executeQuery(sql);

    rs.next();
    int id_col = rs.getInt("ID");
    String id = Integer.toString(id_col);
    String first = rs.getString("First_Name");
    String last = rs.getString("Last_Name");
    String job = rs.getString("Job_Title");

    textID.setText(id);
    textFirstName.setText(first);
    textLastName.setText(last);
    textJobTitle.setText(job);
}
catch (SQLException err) {
    JOptionPane.showMessageDialog(Workers.this, err.getMessage() );
}
```

Run your programme and test it out. You now be able to delete records from your database.

And that's it - you now have the basic knowledge to write a database programme in Java using a GUI. Congratulations, if you got this far!