

Introduction to Netbeans

This document is a brief introduction to writing and compiling a program using the NetBeans Integrated Development Environment (IDE). An IDE is a program that automates and makes easier many tasks that programmers would otherwise have to perform themselves. While many IDEs exist for Java, we will focus only on the NetBeans IDE.

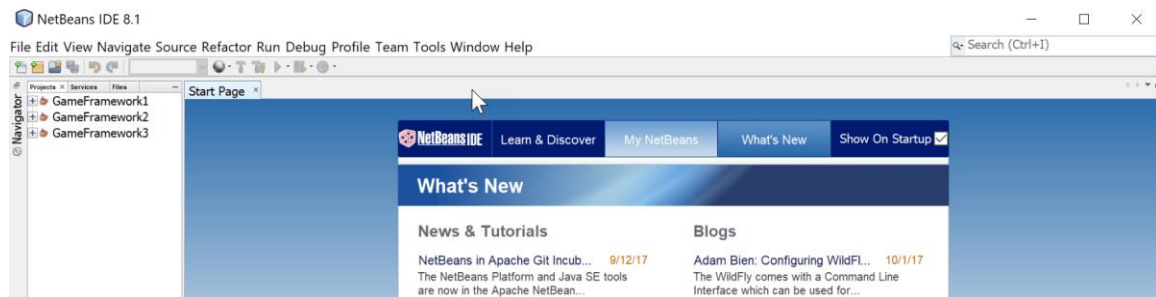
Installation

You can download NetBeans from <http://www.netbeans.org>. There are numerous download options, e.g. including web development, database, etc. The “Java SE” version is sufficient for purposes of this class.

The NetBeans IDE is a big file --- a minimum of around 30 MB. After you have downloaded the file, simply execute the file to install the software.

Starting NetBeans

After NetBeans has started it should bring up the main window that looks something like this:

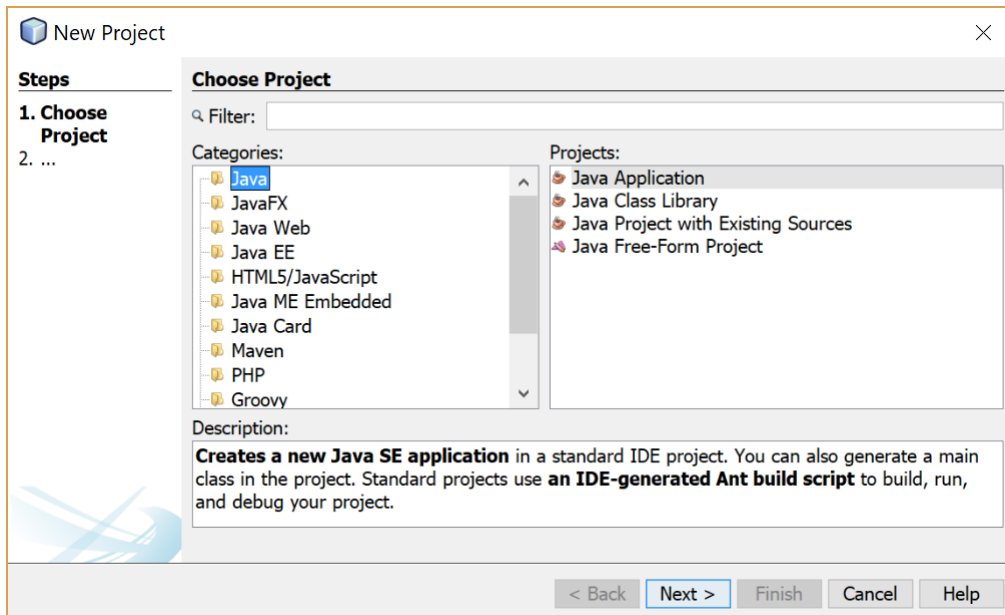


Depending on the version of NetBeans, yours probably won't look exactly the same but should be similar.

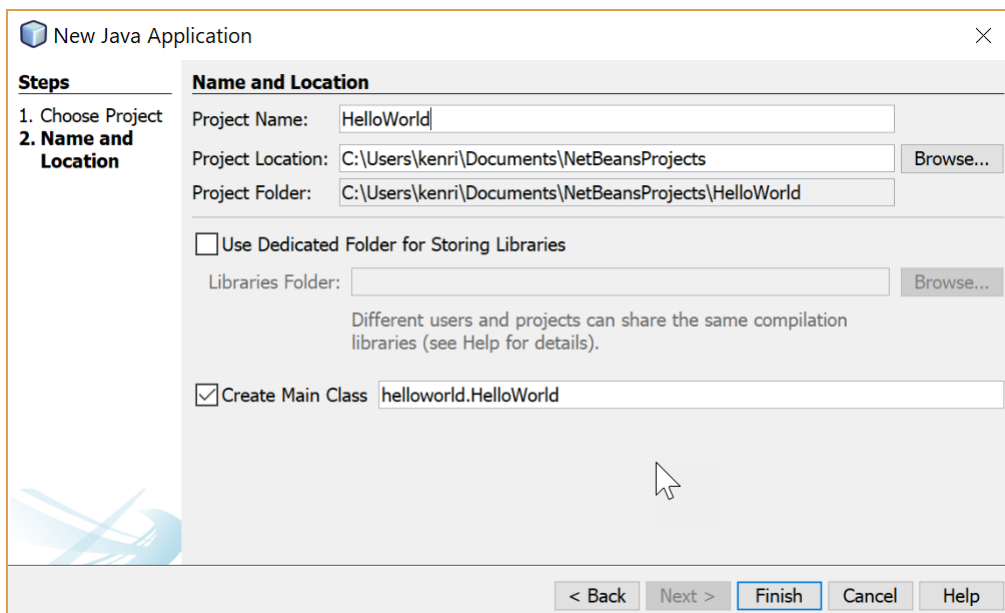
Building a Sample Project – Hello World

Let's start by creating in NetBeans a “Hello, world” program. The process will be similar when working on your own programs.

From the main menu select File and New Project and select Java, Java Application:



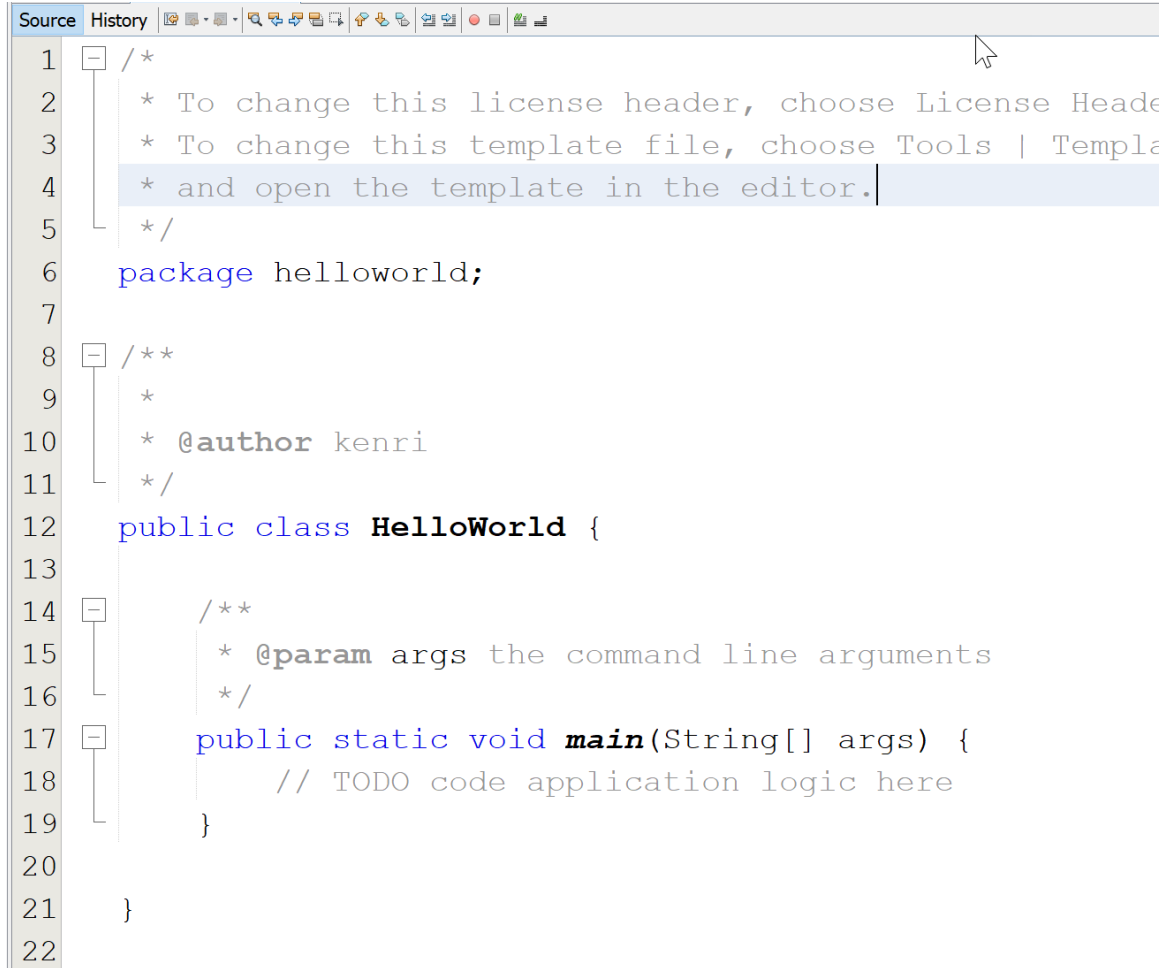
Click Next and then pick a name for your project that is meaningful, like “Homework1Problem2” or such. Note the location where your project will be stored.



Click on Finish and NetBeans will create your project.

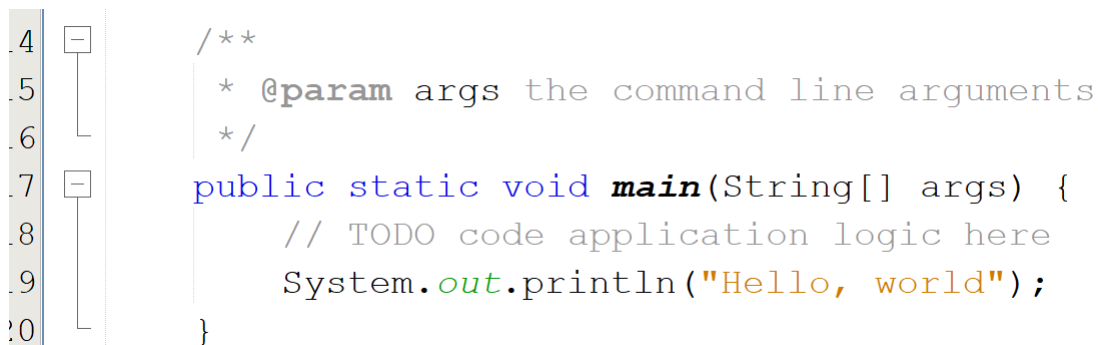
This will create a new Java class called “HelloWorld” and put it in the default package. You might notice from the warning that it is not recommended to put code in the default package. A package is just a way to separate different libraries of code. We’ll see later how to create our own packages and put our code there instead.

NetBeans will generate a default file that looks like this:



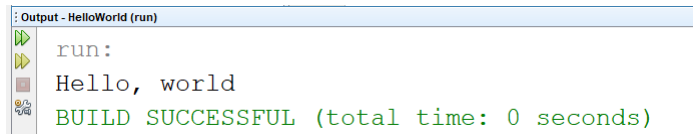
```
1  /*
2  * To change this license header, choose License Headers in Project Properties.
3  * To change this template file, choose Tools | Templates | the file template.
4  * and open the template in the editor.
5  */
6  package helloworld;
7
8  /**
9   *
10  * @author kenri
11  */
12  public class HelloWorld {
13
14      /**
15       * @param args the command line arguments
16       */
17      public static void main(String[] args) {
18          // TODO code application logic here
19      }
20
21  }
```

We can now start typing in the code from our Hello World example and inserting it into the document. You should not type anything before the line with “package”. That has to be the first line in the program.



```
4      /**
5       * @param args the command line arguments
6       */
7      public static void main(String[] args) {
8          // TODO code application logic here
9          System.out.println("Hello, world");
10     }
```

To compile and run your program click on the green triangle “Play” button:



```
Output - HelloWorld (run)
run:
Hello, world
BUILD SUCCESSFUL (total time: 0 seconds)
```

Congratulations, you have just created your first project in NetBeans! You should follow a similar process when working on programs assigned as homework.

If you are going to turn in your project then the easiest way is to locate the project folder (in this example, “HelloWorld” located in “NetbeansProjects”) and then compress the entire folder into a zip file. You can then send the zip file. Note that in the future some instructors may only want your java source files (.java located in the src subdirectory). Sending the entire project folder includes NetBeans gunk that some may not care for if they don’t use NetBeans.

For those that do have NetBeans though, compressing the whole folder is much easier because it includes the NetBeans project and other meta-data used to create the project.

If you want to work on the project at a later date, simply restart NetBeans and all of your files should be visible. If you start to work on many files you may wish to delete some of the .class files or copy out old folders that you no longer use. In particular, the .class files can take up a lot of space.

We have only touched on the basics of using NetBeans here; feel free to explore on your own the many other options that are available. In particular, you may notice that NetBeans will detect many errors as you are typing them. This can be quite helpful in avoiding many common problems.

Debugging

Some have said that any monkey can write a program – the hard part is debugging it. While this is somewhat oversimplifying the difficult process of writing a program, it is sometimes more time consuming and frustrating to debug a program than it was to write it in the first place. However there are tools to help you! The purpose of this lecture is to introduce you to some of these tools.

In this lecture we’ll look at two methods of debugging:

1. Adding print statements, which has the benefit of working in any programming language and in any environment, but is very tedious and can sometimes actually lead to errors
2. Using the debugging utilities in NetBeans (similar tools exist in most IDE’s)

There is also a debugger in Unix called **jdb** for the Java environment. We won’t cover it here, but you should be aware that it exists if you plan to do more development under Unix.

Debugging with Print Statements

You have probably already used this method for debugging. The basic idea is very simple. If your program is not working correctly, try inserting print statements (in Java, this would be `System.out.println` statements) to narrow down where the error is. The print statements could be used to locate the section of code that contains the problem, and they might output the values of some variables that would aid in debugging.

As one example, consider the program below:

```
class DebugTest {
    public static final String STR = "some string here";
    public static void main(String[] args) {
        String s;
        s = STR.substring(0,3);
        // Do some processing of the string
        s = STR.substring(1,5);
        // Do some processing of the string
        s = STR.substring(8,3);
        // Do some processing of the string
        s = STR.substring(22,4);
        // Do some processing of the string
        s = STR.substring(2,3);
        // Do some processing of the string
        return;
    }
}
```

Upon running this program, it crashes and aborts. Something is wrong! (Of course, this program doesn't do much anyway, but it's just an example :)

Fortunately, Java usually tries to give a message about what went wrong. Sometimes this message is useful, and sometimes it isn't. In this case the message is quite useful:

```
C:\homeworks>java DebugTest
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
String index out of range: -5
    at java.lang.String.substring(String.java:1525)
    at DebugTest.main(DebugTest.java:10)
```

This tells us that something is wrong on line 10 That line is:

```
s = STR.substring(8,3);.
```

String index out of range: -5 is a bit cryptic, but the error here is that the parameters are invalid. The first parameter is the index of the start character, while the second parameter should be the index of the end character. However, since 3 is five less than 8, we get an error. Most likely the programmer made this error thinking that the first parameter was

the index of the start character, and that the second parameter was the number of character to include as the substring. (This is the way substring works in C++, so it is not an uncommon error for programmers that learned C++ first!)

In this case, the Java interpreter told us where the error was. In many cases we don't know exactly where we are going wrong. Consider the program below, which tries to convert Fahrenheit to Celsius:

```
import java.util.Scanner;
class DebugTest {
    public static void main(String[] args) {
        int fahr, celsius;
        Scanner keyboard = new Scanner(System.in);

        System.out.println("Enter temperature in Fahrenheit.");
        fahr = keyboard.nextInt();
        celsius = 5 / 9 * (fahr - 32);
        System.out.println("The temp in Celsius is " + celsius);
    }
}
```

When run, it compiles and executes but gives incorrect outputs. For example, on an input of 100 F, we get 0 C, which is incorrect. What is wrong?

To figure out what is going wrong, we could start adding some print statements:

```
import java.util.Scanner;
class DebugTest {
    public static void main(String[] args) {
        int fahr, celsius;
        Scanner keyboard = new Scanner(System.in);

        System.out.println("Enter temperature in Fahrenheit.");
        fahr = keyboard.nextInt();

        System.out.println("You entered: " + fahr);
        System.out.println("Conversion factor = " + (5 / 9));
        System.out.println("fahr - 32 = " + (fahr - 32));

        celsius = 5 / 9 * (fahr - 32);
        System.out.println("The temp in Celsius is " + celsius);
    }
}
```

Recompile and running the new program yields:

```
Enter temperature in Fahrenheit.
100
You entered: 100
Conversion factor = 0
fahr -32 = 68
The temp in Celsius is 0
```

The Conversion factor is obviously incorrect! This should give you enough information to see that the division was incorrectly performed as an int and truncated to 0, since both 5 and 9 are integers by default .

The fix is to make one of these a double and typecast back to an int:

```
celsius = (int) (5.0 / 9 * (fahr - 32));
```


We now get the correct answer, thanks to a little deductive work from our print statement.

The approach of adding print statements has the nice benefit that it works in any programming environment. As such, it is useful to know. However, there are definite drawbacks. First, it is very tedious. The programmer must first figure out where to add the print statements, and then add them and recompile the program. After running the program, more print statements may likely be needed, so they will need to be added and recompiled again. Finally, when the program works as intended, the programmer must go back and remove all of those print statements that were added. Sometimes the process of adding and removing print statements can mess up the program if the programmer deletes an actual line of text by mistake, or if the programmer adds new temporary variables or logic to aid in debugging!

There must be a better way, and there is: the solution is to use a debugger.

The NetBeans Debugger

Users of NetBeans, or virtually any IDE for that matter, have the benefit of a nice, graphical-based debugging environment. The debugger will allow us to set breakpoints, view the contents of variables, and trace through a program one line at a time in a nice GUI environment.

To invoke the debugger, compile the program and then run it in debug mode by selecting D)ebug, Debug Project, or hit the  icon.

NetBeans programs run in one of three modes – design mode, run mode, or break mode. Design mode is where you design and write the code for the program. Run mode is when you run the program. Break mode is when you pause the program to debug it.

If we return to the original program with the bugs, one way to enter break mode is to add a breakpoint. A breakpoint stops execution at a particular line of code and enters Break mode. This is useful when you know that a particular routine is faulty and want to inspect the code more closely when execution reaches that point.

To set a breakpoint, click in the border to the left of the code. A red square will appear. Click the same square to turn the breakpoint off.

```
int fahr, celsius;
Scanner keyboard = new Scanner(System.in);

System.out.println("Enter temperature in Fahrenheit.");
fahr = keyboard.nextInt();

celsius = 5 / 9 * (fahr - 32);
System.out.println("The temp in Celsius is " + celsius);
```

When we run the program and reach this code, the program automatically enters Break mode and stops. Execution stops **before** the line with the breakpoint is executed. The current line is indicated by a green arrow and green highlight:

```
System.out.println("Enter temperature in Fahrenheit.");
fahr = keyboard.nextInt();

celsius = 5 / 9 * (fahr - 32);
System.out.println("The temp in Celsius is " + celsius);
```

The first thing we can do is inspect the value of variables. One way to do this is to hover the mouse over the variable or constant, and a popup window will display its contents:

```
System.out.println("Enter temperature in Fahrenheit.");
fahr = keyboard.nextInt();
fahr = (int) 100

celsius = 5 / 9 * (fahr - 32);
System.out.println("The temp in Celsius is " + celsius);
```

In the above screenshot I have hovered over the “fahr” variable and the pop-up shows that its value is 100. You can also highlight some code and the evaluated expression of that code will be display. In this case I highlighted the “5 / 9” code and it shows the result to be zero:

```
System.out.println("Enter temperature in Fahrenheit.");
fahr = keyboard.nextInt();
5 / 9 = (int) 0

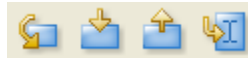
celsius = 5 / 9 * (fahr - 32);
System.out.println("The temp in Celsius is " + celsius);
```

This by itself would give us enough information to debug the program. Note that we did not have to add any print statements!

We can also immediately see the contents of all the active variables by looking in the “Local Variables” tab at the bottom of the screen:

:Watches			:Call Stack			:Local Variables		
Name	Type	Value						
args	String[]	#142(length=0)						
keyboard	Scanner	#143						
fahr	int	100						

We can also step through the program one line at a time using the buttons:



These buttons are used respectively to **step over** a subroutine, **step into** a subroutine, or **step out** of a subroutine. The last button is used to run to the cursor location. We can use these buttons and view our variables change as we run the program. When we define our own subroutines this will make more sense, but for now the first two buttons do the same thing when we're executing code within a subroutine (i.e. method).

Click on the "Step Into" or "Step over" buttons in our example and we get:

```

 celsius = 5 / 9 * (fahr - 32);
 System.out.println("The temp in Celsius is " + celsius);

```

:Watches			:Call Stack			:Local Variables		
Name	Type	Value						
args	String[]	#142(length=0)						
keyboard	Scanner	#143						
fahr	int	100						
celsius	int	0						

Here we can see that the Celsius variable was just set to 0.

As a shortcut, F7 steps into a method call, and F8 steps over a method. These commands are the same for non-subroutines (i.e. the move to the next statement).

If you want the program to just resume execution, click the continue button:

Whenever you are done debugging your program, you must make sure that the debugging session is ended before you go back to edit your code. Click the "Finish Debugging" button to exit the debugger. Then it is safe to edit your code and try again with your changes.

The last debugging option you might find useful is "Run To Cursor". If you are at the point where the program is stopped at a breakpoint or step by step execution, then this

option will treat the cursor option like a temporary breakpoint and execute everything until the program reaches the line where the cursor is.

There are many more commands available in the NetBeans debugger, we have only covered the very basics here. However, what we have covered is sufficient to track down many bugs. In particular, setting breakpoints, tracing through the program, and viewing the contents of variables is the most common task that will aid you in tracking down bugs. Feel free to check out the help topics within the program to learn more. Once you have learned to use the debuggers you will have a powerful tool in your arsenal to write programs more efficiently and bug-free.