1 Hello World in Eclipse

In this first step we are going to write and run our first Java application in Eclipse. After this lecture, you should now be able to:

- Create new Java projects in Eclipse
- Create new Java programs in Eclipse with a simple main method (but not much else yet)
- Run these Java programs

After we have learned how we write and start these very simple applications, we will spend more time on the language and how we write more complex programs.

1.1 Eclipse

Eclipse is a development environment. It started originally as a pure Java development tool, but evolved into a technology platform into which many other tools and applications can be integrated. The development of Eclipse is managed by the Eclipse Community[1], which also hosts other open-source projects, all more or less related to Eclipse.

Understanding all features of Eclipse will take more time than we have. In the following, we only focus on the most important ones to get started with Java programming. There are many sources to learn more about Eclipse. Here are some that can give you a further introduction:

- Eclipse Video Tutorials[2] by Mark Dexter.
- Eclipse And Java For Total Beginners Companion Tutorial Document[3] (PDF) by Mark Dexter
- Introduction to Eclipse[4] (PDF, slides) from UMBC Maryland

Eclipse is itself a Java program. It is also possible to develop Java without a development environment, just using a simple text editor and the compiler from the command line. But since it is much easier to get started in a development environment, we start with Eclipse right away. There are also other development environments for Java, for instance Netbeans[5] and IntelliJ IDEA[6].

[1] http://www.eclipse.org/
[2] http://eclipsetutorial.sourceforge.net/index.html
[3] http://eclipsetutorial.sourceforge.net/Total_Beginner_Companion_Document.pdf
[4] http://www.csee.umbc.edu/courses/undergraduate/341/fall08/Lectures/Eclipse/intro-to-eclipse.pdf

1.2 The Eclipse Workbench

After you installed Eclipse and start it for the first time, you will see the welcome page. You can close this page in the upper right corner, with the button labelled “Workbench”.

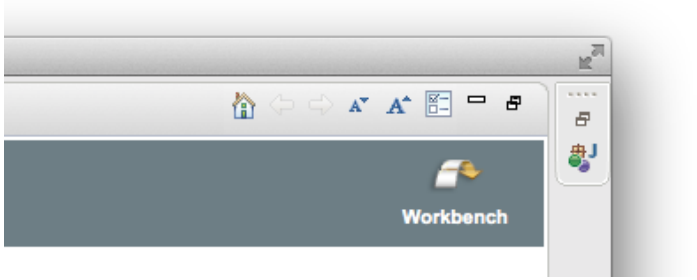

Figure 1: Closing the Welcome Page

The Eclipse **workbench** is the set of windows where you are going to work. The main window is usually the **editor**, where you edit files. There are different types of editors that are specialized for different types of files. The other windows are called **views**. These views show additional information, for instance an overview of your files or a console.

The views and editors are arranged in a particular way that supports a certain task. This arrangement is preconfigured, and these configurations are called **perspectives**. For programming in Java, open the **Java perspective**. You can see in the upper right corner which perspective is currently selected.

If the Java perspective is not already opened, you can open it:

Window / Open Perspective / Other... / Java

You can re-arrange the views and editors as you wish. If you want to go back to the orignal layout of a perspective, simply select *Window*

[5] https://netbeans.org/
[6] http://www.jetbrains.com/idea/

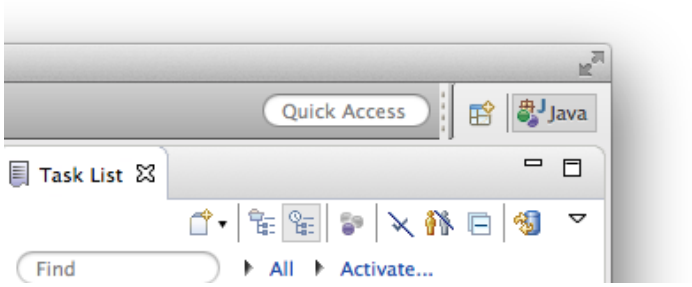

Figure 2: The Java Perspective

/ Reset perspective... You can also open more views via *Window / Show View / Others...*

The Java perspective contains by default a number of views that are useful for Java editing. For now, we are not going to use the Task list, Outline, Declaration and Javadoc views. To get more space on your screen, you can close them. The only views we are going to use are the following:

- The **Package Explorer.** It shows the Java source files and important information about our Java project.
- The **Java Editor.** This is where we are going to write our Java source code.
- The **Problems view** displays explanations of programming errors.
- The **Console view** shows the output of Java programs when they run. This view will open automatically when you run a Java program from within Eclipse.

1.3 The Eclipse Workspace

Eclipse organizes all your files in the **workspace**. This workspace corresponds to a folder on your hard disk, and Eclipse asks you at startup which workspace it should open. This means that you can have several workspaces. I have, for instance, a speparate workspace for this course (ttm4175), but a different workspace for the course ttm4115.

The workspace contains several Eclipse **projects**. These projects correspond to sub-folders within the workspace folder. (Projects can also be configured to reside in totally different places, but we ignore that for now.)

Eclipse projects are units of work that belong together, but they can still refer to each other. When you develop a large application, for instance, you can structure it into different projects that refer to each other. Later in the course, for instance, we will have different Eclipse projects for each larger exercise, just to separate things from each other. You can see the projects in the Package Explorer. It will probably be empty; so let's create a project first.

When you create a new Eclipse project, you can select the type of project. When we select to create a Java project, for example, Eclipse will already create some folders and files that are useful for Java development. It will also create some default settings. In addition, the type of project determines what happens in the background whenever you save a file, but more on that later.

- Create new projects via File / New Project...
- From there, select **Java Project**.
- Give the project a name, for instance *ttm4175*
- The project will be created in the default location (the folder within the workspace folder on your hard disk) and with the default settings for Java projects. These defaults are good for us, so leave them as they are and select **Finish**.

(In the screenshot, you see that for the JRE, Java 8 is selected. Since we ask you to install Java 7, this will be different in your case. Simply leave the JRE value at its default, for the Hello World it will work anyways.)

1.4 The Java Project and the Package Explorer

After creating the Java project, it will show up in the Package explorer view.

The source folder, called *src* will contain all our Java source files. Since we have not created any source code yet, it is empty. So let's create a new class:

- Right-click the *src*-folder
- Select New / Class

The class needs a name. We call it *HelloWorld*, and enter it in the field labelled *Name:*. The package in which we are going to

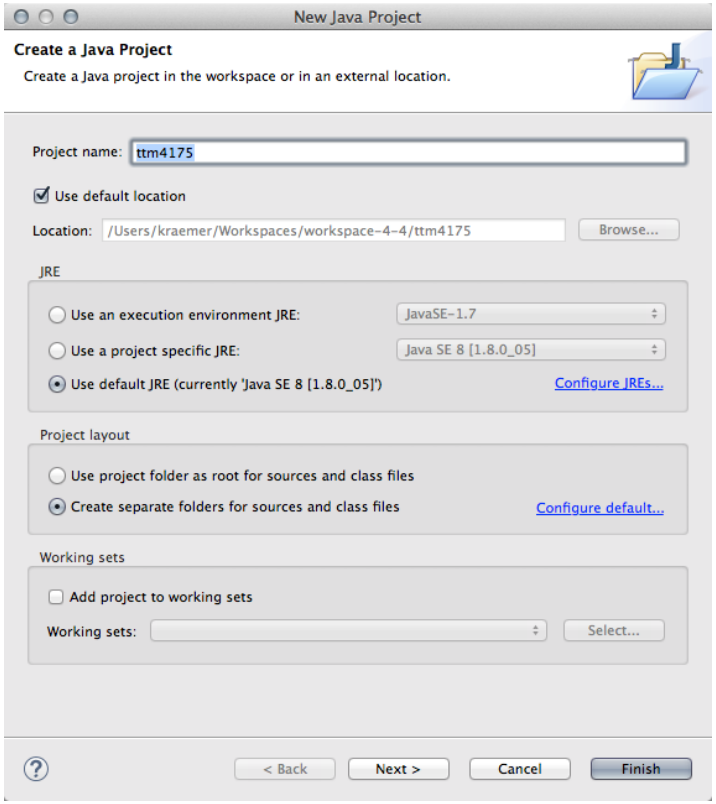

Figure 3: New Project Wizard

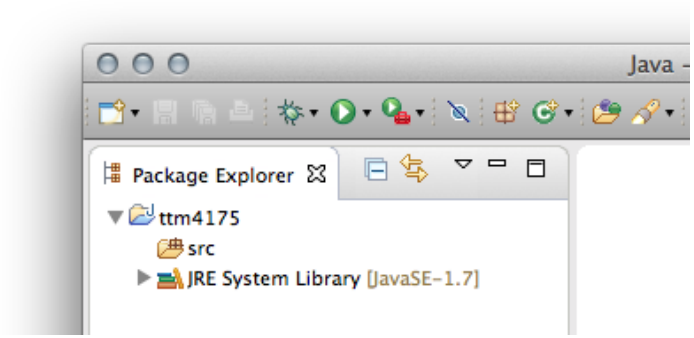

Figure 4: New Project in the Package Explorer

create the class is already called the same as the project, that is *ttm4175*.

Since we want to execute that class, we also need a main method. Therefore, we click the check box *public static void main(String[] args)* (Later more on what that is.) The other settings are good as they are.

Click finish, and the following class will be created:

1.5 Hello World

To write our program, we remove the comments (the lines that are started by the comment markers "//"), and replace it with a code line that prints out a message:

```
public static void main(String[] args) {
    System.out.println("Hello World!");
}
```

(We will later have a close look at all the elements in that code line.)

1.6 Running Hello World

In Eclipse, we can directly run Java programs:

- Select the file *HelloWorld.java* in the Package Explorer
- Select Run As / Java Application

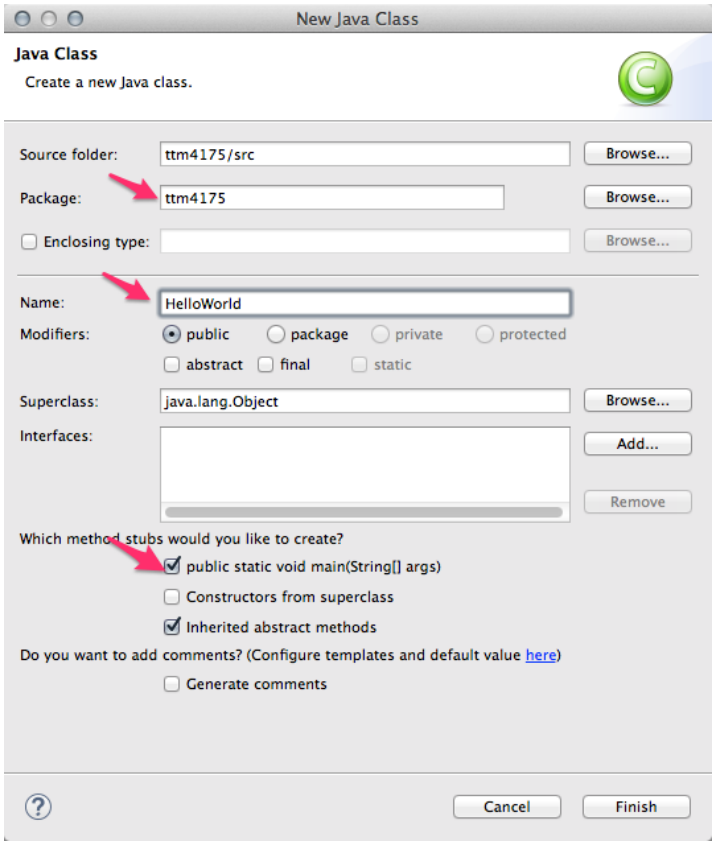

Figure 5: New Class Wizard

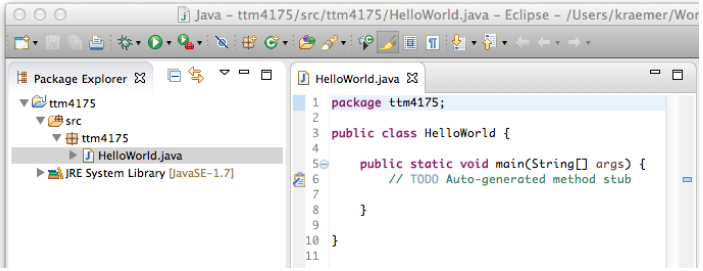

Figure 6: Java Editor with the New Class

The console view will open, and you will see the following output:

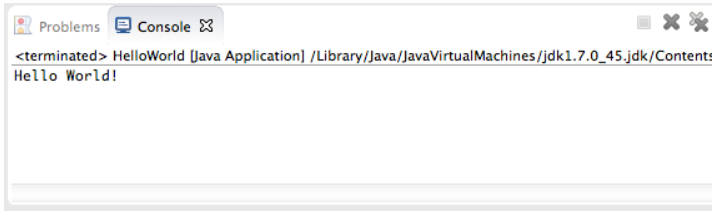

Figure 7: Console with Output for Hello World

We have just created and started our fist Java application!

1 The Java Language - Part 1

In this lecture, we will learn some parts of the Java language to program very simple applications.

1.1 The Main Method

Whenever we want to create a program, it needs to be declared into something that is called a class. This class is defined in a file that has the same name as that class. For now, we call that class `Main`. The entire file will then look as the following:

```
package ttm4175;

public class Main {

    public static void main(String[] args) {

    }

}
```

As you see, we have also declared a package, called `ttm4175`. Packages are simply a way to group related classes together. All Java classes should be sorted into a package.

To keep package names unique, they are sometimes built up like web addresses. For instance, there is the Apache Camel project. Code within that project is organized as packages that start with `org.apache.camel`.

Java itself defines packages that contain classes. These packages start with `java`. For instance, `java.utils` contains a lot of code for lists and other data structures. (So packages do not *need* to correspond to web addressed.)

The name of our class is `Main`. With *fully qualified name* we refer to the name of the class with the package added as prefix. The fully qualified name of our class is `ttm4175.Main`.

The packages also organize how the Java file is stored. The class with the fully qualified name `ttm4175.Main` must be stored within a file called `Main.java`, and stored in a folder called `ttm4175`. If it is stored somewhere else, we will not be able to execute it.

Whenever a Java program is started, the virtual machine executed it main method. This method is declared in the following way:

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

This may look overwhelming at the first glance. Here is what the different keywords and letters mean:

- Between the curly braces {...} resides the method body. It contains all the logic that is executed when the method is called.
- **public** means that this method can be called also from code outside the package declaration of this class. When the virtual machine calls this method, it calls it from outside of the package.
- **static** means that we can call this method of class Main without first creating a new object of class Main.
- **void** means that the method does not return any value.
- **String[] args** is the parameter that is passed to the method. String is a data type, and the square brackets mean that the argument is an array of strings. Within the body of the method, we can refer to the arguments by the name **args**.

All of these elements will get clearer when we look in more detail to the different concepts.

1.2 Back to Hello World

We can print the message *Hello World!* to the console with the following statement, which we add into the main method:

```
public static void main(String[] args) {
    System.out.println("Hello World!");
}
```

- **println** is the name of the method that we need to call.
- This method is defined as part of a class. This class is made available via a variable *out* that is declared within the class *System*. Therefore, we call the method with **System.out** as prefix.
- The simple brackets **()** are part of the method call and embrace any arguments that we pass to the method.
- **"Hello World!"** is the declaration of the String that we want to print out. Everything between the quotation marks (" ") is part of that string.

– The semicolon ; is used after most statements. It can be tricky to place them correctly when you begin learning Java, but after a while you will not have a problem with it. If you want to know more about the semicolon, read this blog post[1] explaining a bit more.

1.3 Variables and Primitive Types

The types byte, short, int, long are representations of signed numbers. *Signed* means that they can be negative or positive. The different types differ in the number of bits that are used to represent them. Bytes have 8 bits, and they can represent values from -128 to 127. Shorts have 16 bits, and can therefore range from -32,768 to 32,767. Integers have 32 bits, and range from -2^31 to 2^31-1. Longs have 64 bit, and range from -2^63 to 2^63-1.

The types float and double are used to represent numbers with a floating point.

Chars represent letters, like `a` or `*`. They are a 16-bit Unicode character.

You can learn more about types here[2].

Let's run the following code:

```
char letter = 'a';
System.out.println("Letter: " + letter);

int number = 2;
System.out.println("Number: " + number);

boolean flag = true;
System.out.println("Flag: " + flag);
```

The output is like this:

```
Letter: a
Number: 2
Flag: true
```

[1] http://beginwithjava.blogspot.no/2008/06/those-pesky-semicolons.html
[2] http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

1.4 Strings

Like the primitive types above, Strings are fundamental types that we need in most applications. Strings are sequences of chars, and used to store text of all sorts. Technically, they are not primitive types, but defined by the class `java.lang.String`. That means that when we work with strings, we actually work on objects. But more on that later.

We can declare a variable for a String like this:

```
Sting hello = "Hello";
```

Here we already assign a value to it, it is the word *Hello*. We have already seen how to print a string to the console:

```
System.out.println(hello);
```

Note that we print here the value within the variable `hello`.

But let's do some more interesting stuff:

```
// add two numbers together
int a = 5;
int b = 3;
int c = a + b;
String result = "Adding " + a + " and " + b + " is " + c;
System.out.println(result);
```

The output:

```
Adding 5 and 3 is 8
```

What's happening here: We declare two integer variables, a and b. Then we declare a third variable, c, and assign it the value a+b. So for numbers, '+' really means addition. After that, we create a string named result. This string is created from several smaller strings, which are combined with the '+' operator. For strings, the '+' means concatenation. The result is a single string that we print out.

1.5 Arrays

Whenever we want to work on several values of the same time, we can use an array. An array can hold a fixed number of values. For instance, we can create an array of integers:

```
int[] myIntegers;
```

At this point, we have only declared that the variable `myIntegers` will hold an array of integers. We have not yet created the array. To create it, we use the following statement:

```
myIntegers = new int[10];
```

When we create the array, we have to declare how many elements the array should have. This is the length of the array. Once the array is created with a certain length, it cannot be changed.

All the elements of the array have an index number. **The index starts with 0.** So, the first element of the array has index 0. And if the array has 10 elements, the index of the last element is 9.

To set the values of certain elements, we refer to their index:

```
myIntegers[0] = 100;
myIntegers[9] = 900;
```

We can access the values in a similar way:

```
int x = myIntegers[0];
// x is now 100
System.out.println(x);
```

Let's print all value of the array:

```
System.out.println("Value at index 0 is: " + myIntegers[0]);
System.out.println("Value at index 1 is: " + myIntegers[1]);
System.out.println("Value at index 2 is: " + myIntegers[2]);
System.out.println("Value at index 3 is: " + myIntegers[3]);
System.out.println("Value at index 4 is: " + myIntegers[4]);
System.out.println("Value at index 5 is: " + myIntegers[5]);
System.out.println("Value at index 6 is: " + myIntegers[6]);
System.out.println("Value at index 7 is: " + myIntegers[7]);
System.out.println("Value at index 8 is: " + myIntegers[8]);
System.out.println("Value at index 9 is: " + myIntegers[9]);
```

(We will later see how we do that much more elegant with a loop.)
When we run the program, we get the following output:

```
Value at index 0 is: 100
Value at index 1 is: 0
Value at index 2 is: 0
Value at index 3 is: 0
Value at index 4 is: 0
Value at index 5 is: 0
Value at index 6 is: 0
Value at index 7 is: 0
Value at index 8 is: 0
Value at index 9 is: 900
```

We have only set the first and the last value. The other values are 0, which is the default value of an integer when we do not assign a value to it.

You can learn more about arrays here[3].

1.6 More Stuff to Do

We have seen that we can use System.out.println() to write something to the console. Java comes with many other libraries of classes that can be used for all sorts of things.

```
// get the current time
long milliseconds = System.currentTimeMillis();
System.out.println("Milliseconds since 1 January 1970 00:00:00: "
    + milliseconds);
```

The class Math provides many mathematical operations:

```
// get a random number
double random = Math.random();
System.out.println("A random number between 0.0 and 1.0: "
    + random);
```

1.7 While Loops

Loops repeat the statements that are declared in their body. A while-loop looks like this:

```
while( <condition> ) {
    <statements>
}
```

The following code prints the numbers from 0 to 9:

```
int x = 0;
while(x<10) {
    System.out.println("x = " + x);
    int x = x + 1;
}
```

[3] http://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html

Within the round brackets there is the expression `x<10`. The loop is executed while this statement remains true. Within the curly braces, the statements print out the value of x, and increase x for each repetition.

So the output of this code is the following:

```
0
1
2
3
4
5
6
7
8
9
```

You can learn more about the while loop here[4].

1.8 For Loops

Another loop is the for-loop. Like the while loop, it executes the statements in its body repeatedly. What is different is the control of the loop. The construct looks like this:

```
for( <declaration> ; <condition> ; <statement> ) {
}
```

Following the keyword for are the control sequence...

```
for(int i=0; i<10; i++) {
    System.out.println("i = " + i);
}
```

The for loop can declare a variable, a condition that must hold while repeating, and a statement that is executed each time the loop is executing another repetition.

Let's go back to the array from above. We can also loop over all elements of an array:

```
for(int i=0; i<myIntegers.length; i++) {
    System.out.println("Value at index " + i + " is: "
        + myIntegers[i]);
}
```

[4] http://docs.oracle.com/javase/tutorial/java/nutsandbolts/while.html

With `myIntegers.length`we get the length of the array, which is 10 in the example. We use this as the upper value. (Note that we use `x<myIntegers.length`, this means we do run the loop with `x=9` as last value.)

You can learn more about the for loop here[5].

1.9 If Statements

If statements are for branching depending on a condition. An if statement looks like this:

```
if ( <condition1> ) {
    ...
} else if ( <condition2> ) {
   ...
} else {
   ...
}
```

- The first if-branch must be there.
- There can be any number of else-if branches, which are selected if their condition is true and none of the conditions of the previous above is true.
- The else-branch (without condition) can come as last branch and is optional. It is taken if none of the other branches is taken.

```
double random = Math.random();
System.out.println("A random number between 0.0 and 1.0: " + random);
if(random > 0.8) {
    System.out.println("Random number is larger than 0.5!");
} else if (random <=0.3) {
    System.out.println("Random number is smaller than 0.3!");
} else {
    System.out.println("Random number is between 0.3 and 0.5!");
}
```

You can learn more about if statements here[6]. Another statement that can be used similarly to the if statement is the switch statement[7]: For now, we leave it out.

[5] http://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html
[6] http://docs.oracle.com/javase/tutorial/java/nutsandbolts/if.html
[7] http://docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html

1.10 Operators

Operators can calculate values based on the arguments passed to them.

There are boolean operators, like `&&`and `||` that do a logical `and` and `or` on two boolean variables. We can for instance use the following code to print out a truth table for these operators:

```
boolean a = false;
boolean b = false;
System.out.println("   a     b    a || b");
System.out.println(" " + a + " " + b + " " + (a || b));
b = true;
System.out.println(" " + a + " " + b + "  " + (a || b));
a = true;
b = false;
System.out.println(" " + a + "  " + b + " " + (a || b));
b = true;
System.out.println(" " + a + "  " + b + "  " + (a || b));

a = false;
b = false;
System.out.println("   a     b    a && b");
System.out.println(" " + a + " " + b + " " + (a && b));
b = true;
System.out.println(" " + a + " " + b + "  " + (a && b));
a = true;
b = false;
System.out.println(" " + a + "  " + b + " " + (a && b));
b = true;
System.out.println(" " + a + "  " + b + "  " + (a && b));
```

We have alreay used some operators to compare values. There are the following:

```
x == y    true if x and y are equal
x =! y    true if x and y are not equal
x <  y    true if x is smaller than y
x <= y    true if x is smaller than or equal to y
x >  y    true if x is larger than y
x >= y    true if x is larger than or equal to y
```

There are also math operators that work on numbers:

```
x + y    (plus) the sum of x and y
x - y    (minus) the difference of x and y
x / y    (division) x divided by y
x % y    (remainder) the rest that remains after dividing x by y
```

There are more operators, and the ones above can be used on other types than we have shown in the example. But I recommend that you have a look for the right operator whenever you need it, and learn more about them[8] when you know how you want to use them.

[8] http://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html