

COMP 401
Prasun Dewan¹

1. Conventional Programming in Java for those who know how to program

The main method

Let us start our study of programming by looking at the simplest program we can write in Java:

```
0. package warmup;
1. public class AHelloWorldGreeter
2. {
3.     public static void main (String[] args)
4.     {
5.         System.out.println ("Hello World");
6.     }
7. }
```

This program outputs:

Hello world

This program is more complicated than equivalent programs would be in other programming languages. For instance, in Basic, we would simply write:

```
print "Hello World"
```

This is because of Java's heavy emphasis on modularity, which makes large programs easy to write, but trivial programs such as the one above more complicated than necessary. Imagine a writing style in which you composed a sequence of sentences without grouping them into higher-level units such as paragraphs, sentences, and chapters. Such a writing style is easy to learn, and useful for small compositions, but does not lend itself to easy-to-understand complex compositions. That is essentially the kind of style encouraged by Basic. In contrast, Java provides a much richer set of constructs for grouping instructions, which makes it difficult to learn but suitable for complex programs. Unfortunately, in Java, it is not possible to ignore some of these constructs in a simple program such as our example. Let us try and understand the various parts of it to the extent we can at this early point in the course.

Methods and Arguments

Let us start with line 5, which is the heart of the program:

```
5.         System.out.println ("Hello World");
```

¹ © Copyright Prasun Dewan, 2000.

Its syntax makes it easy to guess that it prints “Hello World”. We can parse it as a verb and an object², with the former specifying an action and the latter the object to be acted on. Or we can use the Math functional notation we are familiar with (e.g. `sqrt(2)`) to parse it as an operation and an operand

Verb	Object
Operation	Operand
Method	Argument
System.out.println	("Hello World")

Figure 1 Parsing a Method Invocation

In the world of Java (and object-oriented programming languages in general), an operation is called a *method*, an operand is called an argument; and performing an operation is called *invoking* a method. Thus, this line invokes the method, *System.out.println*, supplying it the argument “Hello World”.

We shall see later the reason for the long name of this method. For now, we will use the shorthand, *println*, for it.

A method invocation is a step taken by a program to perform its task. While this program consists of a single method invocation, in general, a program may contain several method invocations. Like a complete program, a method can take one or more arguments, which influence the action taken by the method. The *println* method is invoked with one argument, “Hello World”, which identifies the *string* to be printed.

A string is a sequence of characters, which may be displayed as output to the user, and, as we shall see later, received as input from the user. The double quotes indicate the start and end of the string – they are not part of it. Thus, they do not appear in the output shown in Figure 2.

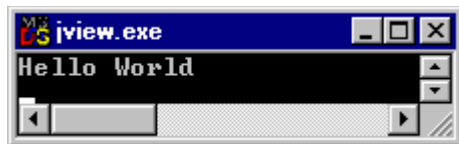


Figure 2 A Transcript Window

For each program, Java creates a separate window for displaying its output and gathering its input, as shown above. We shall call this window the program’s *transcript* window, since it shows a transcript of an interactive session with the program.

As we shall see later, computation, in Java will mainly involve invoking methods. Some of these methods, such as *println* are defined by the language; others are *defined* or *implemented* by the program itself. We shall refer to the former as *predefined* and the latter as *programmer-defined*.

² The use of the word object here should not be confused with its use in object-oriented programming.

Classes

Unlike the case in many other languages such as Pascal and Basic, a method definition cannot appear in isolation—it must be a part of a *class*, which essentially groups together a set of related method definitions. For instance, the definition of the method `println` appears in the class `java.io.PrintStream`. Similarly, the method `main` defined by this program, appears in the class `AHelloWorldGreeter`. By requiring methods to be defined in classes, Java encourages cataloging of methods, an important concern from the point of understanding a program.

Class Definition, Identifiers & Keywords

We are now in a position to understand the whole program.

In general, a Java program consists of several classes. Since this is a simple program, it consists of one class, `AHelloWorldGreeter`.

The class declaration consists of two parts: a *class header* and a *class body*. The class header is the first line of the program, and consists of the part of the class definition before the first curly brace, `{`:

```
1. public class AHelloWorldGreeter
```

The class body is the rest of the program.

```
2. {
3.     public static void main (String args[])
4.     {
5.         System.out.println ("Hello World");
6.     }
7. }
```

The class header contains information about the class that is of interest to the users of the class. As a user of a method defined in a class, all we need to know about the class is its name, which is what the class header tells us.

It consists of two *keywords*. The first is the word, **class**, which is defined by Java, while the second is the class name, which is defined by the programmer. A *keyword* is a predefined Java word that has a special meaning to it. In this case, it tells Java that what follows it is the start of a class definition. A keyword is also called a **reserved word** because it is reserved by the language in that it cannot be overridden by the programmer. We will use **boldface** to identify keywords. Not all predefined Java words are keywords. For instance, *String* and *println* are not keywords, and thus can be overridden by a program. Though, in this class, we will not be overriding predefined Java words, it is important to distinguish keywords from overridable (predefined and programmer-defined) words.

Method Declaration

The class body contains the implementation of the class, which consists of the definition of the methods of the class. In our example, it consists of the definition or declaration of a single *method*, `main`, enclosed within the outermost curly braces (lines 3-6):

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

```

4.      {
5.          System.out.println ("Hello World");
6.      }

```

Like a class declaration, it has two parts: a *header* and a *body*. The method header is the first line of the declaration (line 3 of program), before the curly brace:

```

3.      public static void main (String args[])

```

The method body is the rest of the declaration (lines 4-6):

```

4.      {
5.          System.out.println ("Hello World");
6.      }

```

The method header contains information that is of interest to its users. The user of a method needs to know how to invoke the method. Therefore, the header includes the name of the method and the nature of the argument accepted by it. It consists of three Java *keywords*, **public**, **static**, and **void**, followed by the method name, *main*, followed by an argument specification. The argument specification tells us the argument is a list of strings named *args*. The three keywords indicate various properties of the method. The keyword **public** says that the method is “visible” outside the class to other software modules—in particular the interpreter. **static** says that it is a “class method” rather than an “instance method”. For now, since we are not doing object-based programming, all methods must have this keyword next to them. When we create objects dynamically, we will omit this keyword. **void** says that it is a “procedure” rather than a “function”, that is, it does not return a value.

Let us postpone a more detailed discussion of the method header. For now, it is important to know that it indicates this is the *main* method of the program. (Though this program contains a single method declaration, a Java program can contain many method declarations.) A main method is a special method of a program in that it is automatically called by the Java interpreter when the program is executed. Every program must have a main method, and you should type its header exactly as shown above to indicate that it is the main method. Check that you have done so in case you get an error when you execute the program saying that the main method was not found.

The method body defines what happens when the method is invoked. It consists of the method invocation we saw earlier. Thus, the main method is defined in terms of an existing method, just as the code-word *act1* was defined in terms of existing English words.

Package

Java also supports packages, which are essentially directories. The line:

0. **package** warmup

puts this class in the warmup package. The Borland environment automatically creates a directory in which this class is put. We will put all of the other classes we look at in this chapter in the same package. A class is fully specified by prefixing it with its package name. For instance, the class we defined above is fully specified as: warmup.AHelloWorldGreeter

Programming Process and Program Structure

Thus, we now better understand the sequence of events that occur when the program is executed:

1. The Java interpreter invokes the main method defined in class `AHelloWorldGreeter`.
2. The method executes the single method invocation in its body.
3. This method, in turn, invokes the `System.out.println` method.
4. This method, which is defined in class `java.io.PrintStream`, prints its argument in the transcript window.

This example illustrates the basic process of programming – defining new classes and methods using existing classes and methods provided by the Java language and libraries. In this example, we have defined a simple new class and method to print the specific string, “hello world”, using the existing method `println` of class `java.io.PrintStream` to print general strings.

This example also illustrates the general structure of a Java program, shown in figure 10.

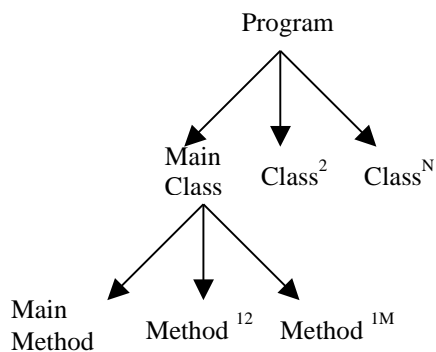


Figure 3: Structure of a Java Program

In general, a program consists of a main class and can have several other classes. The main class must have the main method and can have several other methods. Other classes can have arbitrary methods. In this program, of course, we have seen a very simple example of this structure—one class and one method (Figure 4). Initially we will focus on programs with this structure.

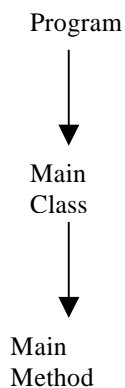


Figure 4 Simply Structured Java Program

Such programs have the syntax described by the following template:

```
public class <ClassName> {
    public static void main (String args[]) {
        ...
    }
}
```

where <ClassName> and the actions taken by the main method are specific to the program you are writing. Such a template, in fact, is automatically generated by some programming environments such as Visual J++ 6.0. For now, use this template for the programs you write.

Developing and Running the Program in a Bare-Bone Environment

We have seen above how the program works but not how it is edited, compiled, or interpreted. The answer depends on the programming environment we use. All Java programming environments provide a Java compiler and a Java interpreter, normally called, *javac* and *java*, respectively. Moreover, the operating system provides a text editor and a command interpreter. The combination of these three tools constitutes a bare-bone programming environment, which is sufficient, but not always suitable, to develop Java programs. In case you will be using the Borland environment directly, you can skip this section.

Let us illustrate how this simple program can be developed using such an environment. You can use the text editor to enter the program text, and save it in a file called `AHelloWorldGreeter.java`. In general, each program class should be saved in a separate file whose name is the name of the class followed by the suffix. `.java`.

Your next steps are to compile and interpret the program. Before we do so, we must put the directory or folder in which the compiler and interpreter reside in the *program search path*, which is a list of folders in which the command interpreter searches for programs you name.

Assuming our path is set, we can simply type the following command to compile the file:

```
javac AHelloWorldGreeter.java
```

This command must be executed in the directory in which the class is defined. In our example, it must be executed in the warmup directory. The compiler creates a file called `AHelloWorldGreeter.class`, which contains the object code for this program. You can now ask the Java interpreter to execute this class:

```
java warmup.AHelloWorldGreeter
```

This program must be executed in the directory containing the package, that is, in the parent of the warmup folder.

At this point, you will see the greeting printed in the transcript window. In general, to execute a program, you must type:

```
java <Full Name of Main Class>
```

The names of the compiler and interpreter depend on the programming environment.

Case matters in the names used in Java. In some programming environments, it also matters in the names supplied to the tools. So if we typed:

```
java warmup.ahelloworldAHelloWorldGreeter
```

you may get an error message saying that the class `ahelloworldAHelloWorldGreeter` was not found.

Program Arguments

Recall that a program can take arguments provided by the user when the program is executed. The following example shows how a program can access an argument entered by the user. It prints the first argument entered by the user in the transcript window.

```
package warmup;
public class AnArgPrinter {
    public static void main (String[] args) {
        System.out.println (args[0]);
    }
}
```

Thus, if you provide the single string as an argument:

```
hello
```

the program will output

```
hello
```

If you provide two arguments:

```
hello world
```

it still prints:

```
hello
```

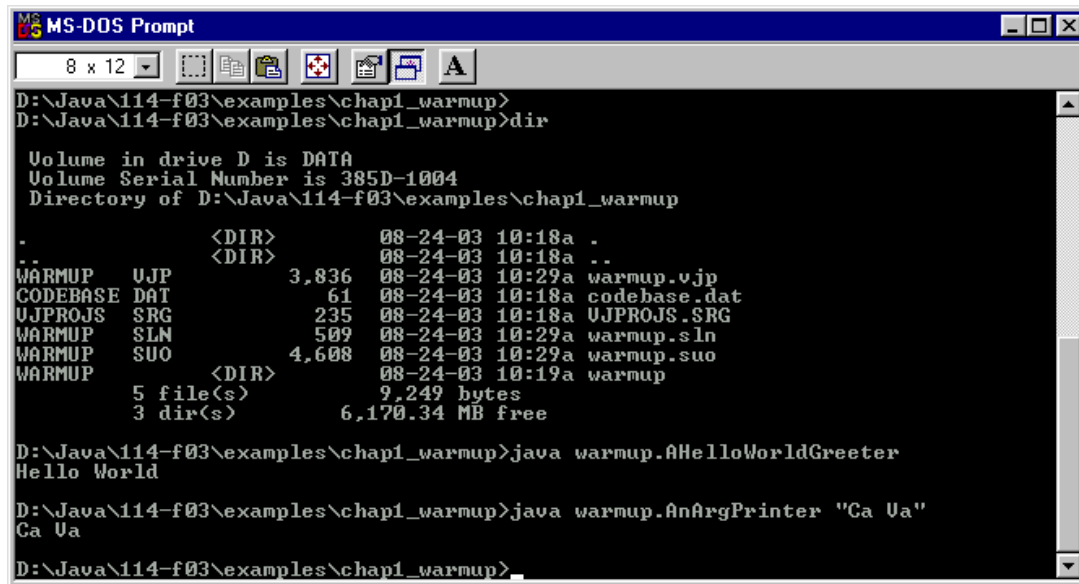
In case you have a multiword string that is to be printed, it is necessary to enclose it in strings, as we did in the first program. Thus, if you specify the argument:

```
"hello world"
```

the output is:

```
hello world
```

To specify arguments from a command window, you need to supply them to the interpreter after the name of the main class, as shown below:



```

MS-DOS Prompt
8 x 12
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>dir

Volume in drive D is DATA
Volume Serial Number is 385D-1004
Directory of D:\Java\114-f03\examples\chap1_warmup

.                <DIR>          08-24-03  10:18a  .
..               <DIR>          08-24-03  10:18a  ..
WARMUP  UJP          3,836  08-24-03  10:29a  warmup.vjp
CODEBASE DAT           61  08-24-03  10:18a  codebase.dat
UJPROJS SRG          235  08-24-03  10:18a  UJPROJS.SRG
WARMUP  SLN          509  08-24-03  10:29a  warmup.sln
WARMUP  SUO          4,608  08-24-03  10:29a  warmup.suo
WARMUP  <DIR>          08-24-03  10:19a  warmup
          5 file(s)          9,249 bytes
          3 dir(s)          6,170.34 MB free

D:\Java\114-f03\examples\chap1_warmup>java warmup.AHelloWorldGreeter
Hello World

D:\Java\114-f03\examples\chap1_warmup>java warmup.AnArgPrinter "Ca Ua"
Ca Ua

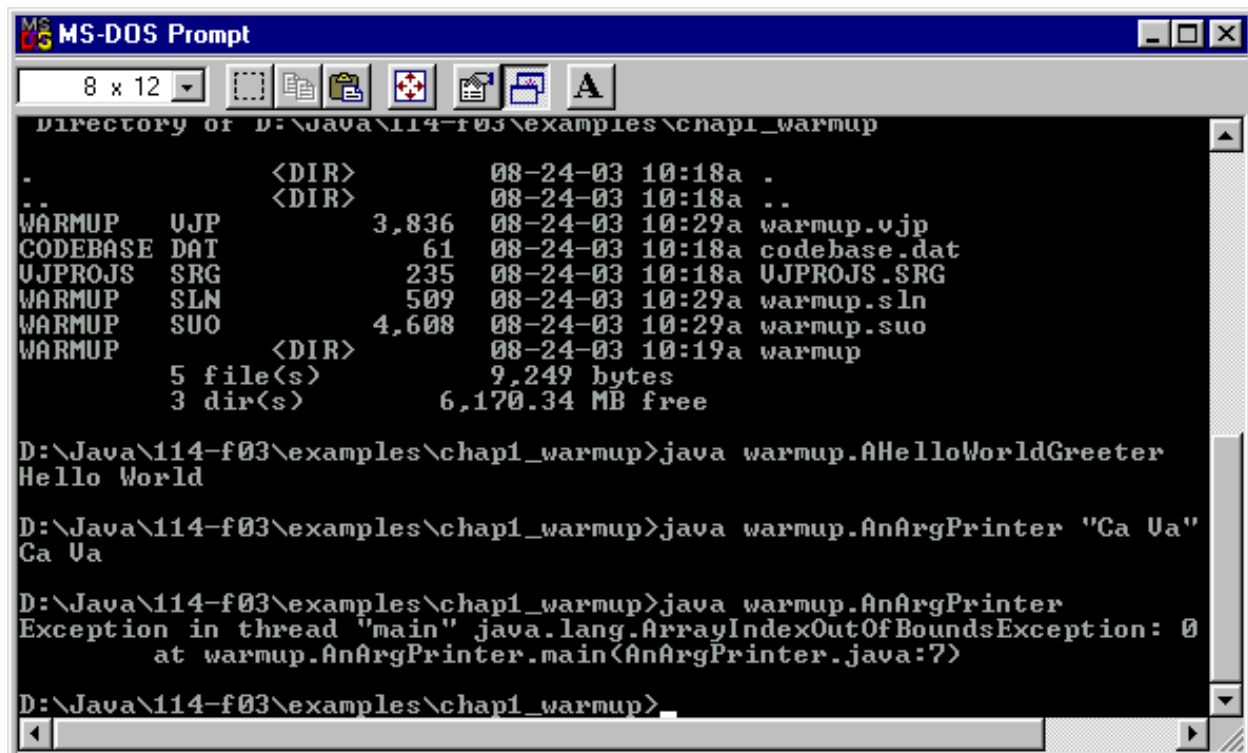
D:\Java\114-f03\examples\chap1_warmup>

```

Figure 5 Supplying Arguments from the Command Window

The arguments specified by a user can be accessed by the main method as `args[0]`, `args[1]`, `args[2]` and so on, where `args` is the name given to the argument list or *array* in the header of the main method. In general, the Nth argument is accessed in the main method as `args[N-1]`.

It is an error to access an argument that has not been entered by the user. Thus, if you supply no argument to the program above, your program will terminate indicating an `ArrayIndexOutOfBoundsException`, as shown in Figure 6.



```

MS-DOS Prompt
8 x 12
Directory of D:\Java\114-f03\examples\chap1_warmup

.                <DIR>          08-24-03  10:18a  .
..               <DIR>          08-24-03  10:18a  ..
WARMUP  UJP          3,836  08-24-03  10:29a  warmup.vjp
CODEBASE DAT           61  08-24-03  10:18a  codebase.dat
UJPROJS SRG          235  08-24-03  10:18a  UJPROJS.SRG
WARMUP  SLN          509  08-24-03  10:29a  warmup.sln
WARMUP  SUO          4,608  08-24-03  10:29a  warmup.suo
WARMUP  <DIR>          08-24-03  10:19a  warmup
          5 file(s)          9,249 bytes
          3 dir(s)          6,170.34 MB free

D:\Java\114-f03\examples\chap1_warmup>java warmup.AHelloWorldGreeter
Hello World

D:\Java\114-f03\examples\chap1_warmup>java warmup.AnArgPrinter "Ca Ua"
Ca Ua

D:\Java\114-f03\examples\chap1_warmup>java warmup.AnArgPrinter
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
    at warmup.AnArgPrinter.main(AnArgPrinter.java:7)

D:\Java\114-f03\examples\chap1_warmup>

```


Figure 6 Array IndexOutOfBoundsException Exception**Conditionals**

The reason for the exception is that the program did not check if the user actually supplied the argument. To do so requires the use of conditionals, illustrated below:

```
package warmup;
public class ASafeArgPrinter {
    public static void main (String[] args) {
        if (args.length == 1)
            System.out.println (args[0]);
        else {
            System.out.println("Illegal no of arguments:" + args.length + ". Terminating program");
            System.exit(-1);
        }
    }
}
```

Here, `args.length` gives the length of the `args` array, and thus, the number of arguments supplied by the user. `System.exit()` terminates the program, the non-zero argument indicating an abnormal termination to the interpreter. The first `+` in the print statement converts the integer on its right to a string and concatenates it to the string on its left. The result string is concatenated by the second `+` to its right string operand.

The output of the program is given below:

```
MS-DOS Prompt
8 x 12
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>java warmup.ASafeArgPrinter "Ca Ua"
Ca Ua
D:\Java\114-f03\examples\chap1_warmup>java warmup.ASafeArgPrinter
Illegal no of arguments:0. Terminating program
D:\Java\114-f03\examples\chap1_warmup>java warmup.ASafeArgPrinter "Ca Ua" "Namas
te"
Illegal no of arguments:2. Terminating program
D:\Java\114-f03\examples\chap1_warmup>
```

The program uses the conditional or if statement. In general, the statement has the syntax:

```

if (<boolean expression>)
    <statement1>;
else
    <statement2>;

```

(Note that parentheses are required around <boolean expression>). What this statement depends on <boolean expression>. If <boolean expression> evaluates to **true**, it executes <statement1>, otherwise it executes <statement2>. <statement 1> is called the *then part* or *then branch*; <statement 2> is called the *else part* or *else branch*; and <boolean expression> is called the *condition* of the if-else

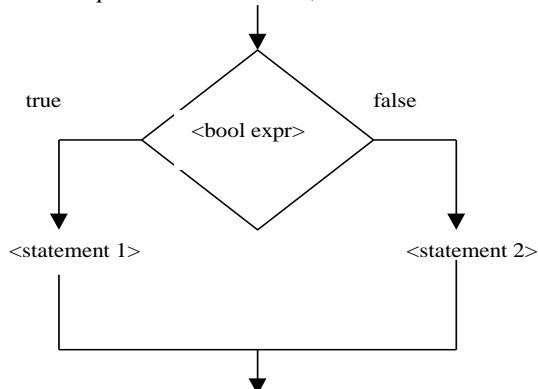


Figure 3 If-Else Statement

Thus, in the method above, the if-else tests the boolean expression:

```
args.length == 1.
```

In case this expression returns true, the statement branches to the then part:

```
System.out.println(args[0]);
```

Otherwise the statement branches to the else part:

```

{
    System.out.println("Illegal no of arguments:" + args.length + ". Terminating program");
    System.exit(-1);
}

```

The else part is a single statement, a statement list also called a compound statement, demarcated by the { and }, which in turn consists of two sub statements, as shown below.

```

if (args.length == 1)
    System.out.println (args[0]);
else
    {
        System.out.println("Illegal no of arguments:" + args.length + ". Terminating program");
        System.exit(-1);
    }

```

Nested Conditionals

Consider the following function, which takes as an argument a score and returns a letter grade based on the score:

```

public static char toLetterGrade (int score) {
    if (score >= A_CUTOFF)
        return 'A';
    else if (score >= B_CUTOFF)

```

```

        return 'B';
    else if (score >= C_CUTOFF)
        return 'C';
    else if (score >= D_CUTOFF)
        return 'D';
    else
        return 'F';
}

```

Figure 7 Branching in the else Part

The use of the if else statement here might seem at odds with the syntax we gave:

```

if (<boolean expression>)
    <statement1>
else
    <statement2>

```

In fact, you might think of it as a special kind of statement in which the keywords **else if** replace the keyword **else**. But actually, it is simply a sequence of *nested conditionals*, that is, if-else statements whose else or then branches contains other conditionals (Figure 5). The following format clarifies this:

```

if (score >= A_CUTOFF)
    return 'A';
else
    if (score >= B_CUTOFF)
        return 'B';
    else
        if (score >= C_CUTOFF)
            return 'C';
        else
            if (score >= D_CUTOFF)
                return 'D';
            else
                return 'F';

```

Since this format takes more space, we prefer the previous one, in which we used the same indentation for the enclosing and enclosed conditionals.

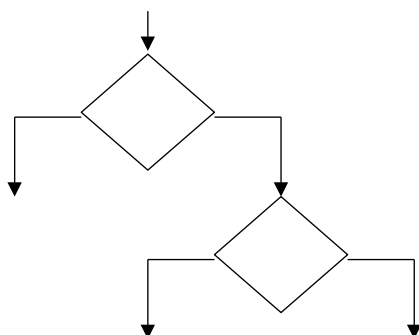


Figure 5 Nesting in the else parts

if Statement

Sometimes we wish to take no action in an else branch. In this case, the *if statement* can be used instead of the if-else statement. It has the syntax:

```
if (<boolean expression>
    <statement>;
```

Example:

```
if (args.length == 1)
    System.out.println ("args[0]");
```

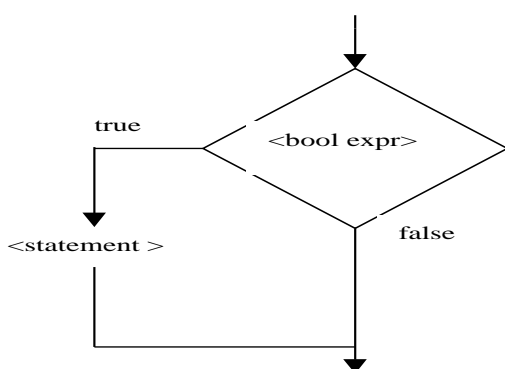


Figure 4 if Statement

We can always use an if-else statement instead of an if statement by putting nothing (the "null" statement) in the else part:

```
if (args.length == 1)
    System.out.println ("args[0]");
else;
```

But the if statement is more elegant when the else is a null statement. Beginning programmers, used mainly to the if-else statement and unfamiliar with the null statement, tend to sometimes put a spurious assignment in the else part:

```
if (args.length == 1)
    System.out.println ("args[0]");
else i = i;
```

Like the previous solution, this is correct but even less elegant!

We shall refer to if-else and if statements as *conditionals*, because their execution depends on some condition. An if-else statement is used to choose between the execution of two statements based on the condition; while an if statement is used to determine whether a statement should be executed based on the condition.

Dangling Else Problem

Consider the following nested if statement:

```
if (score >= B_CUTOFF)
    if (score >= A_CUTOFF)
        System.out.println ("excellent!");
else
    System.out.println ("bad");
```

Which if statement does the else statement match with? If you typed this text using some programming environments, this is how it would be automatically formatted. This may lead you to believe that it matches

the outer if statement. But you could manually format it as:

```

if (score >= B_CUTOFF)
    if (score >= A_CUTOFF)
        System.out.println ("excellent!");
    else
        System.out.println ("bad");

```

Now it looks as if it matches the inner if statement. Our syntax rule of an if-else statement allows for both interpretations. This ambiguity is called the *dangling else* problem. It is resolved by making an else match the closest if. So, in this example, the second interpretation is the correct one. The possibility for confusion here is another reason for not nesting in a then branch.

Loops

Suppose we wanted to print all the arguments a user inputs, as shown in the figure below.

```

MS-DOS Prompt
8 x 12
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>java warmup.AnArgsPrinter "Ca Ua" "Namaste"
Ca Ua
Namaste
D:\Java\114-f03\examples\chap1_warmup>java warmup.AnArgsPrinter
D:\Java\114-f03\examples\chap1_warmup>java warmup.AnArgsPrinter "Ca Ua"
Ca Ua
D:\Java\114-f03\examples\chap1_warmup>

```

The following program uses the while statement to do so:

```

package warmup;
public class AnArgsPrinter {
    public static void main(String[] args) {
        int argNo = 0;
        while (argNo < args.length) {
            System.out.println(args[argNo]);
            argNo++;
        }
    }
}

```

}

The while statement looks, syntactically, much like the if statement (without an else). Recall that an if statement has the syntax:

```
if (<boolean expression>)
    <statement>
```

A while statement has the syntax:

```
while (<boolean expression>)
    <statement>
```

As in the case of an if conditional, <statement> can be any statement including a compound statement. Moreover, like an if conditional, a while loop executes <statement> in case <boolean expression> is **true**, and skips it otherwise. Unlike the former, however, after executing <statement>, it checks <Boolean Expression> again and repeats the process until <boolean expression> is **false**. <statement> is called the while *body* and <boolean expression> the *condition* of the loop. Each execution of the loop body is called an *iteration* through the loop.

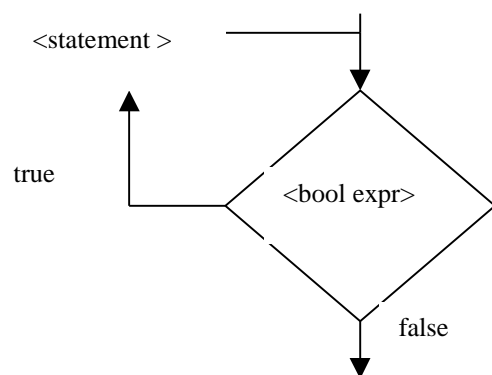


Figure 8 The while loop

Scanning Problem

To gain more experience with loops and introduce other concepts, let us consider the following problem:

```

MS-DOS Prompt
8 x 12
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>java warmup.AnUpperCasePrinter "John F. Kennedy"
Upper Case Letters:
JFK
D:\Java\114-f03\examples\chap1_warmup>

```

This is an example of an important class of such problems: the task of scanning a string of characters. This kind of problem arises in a variety of applications. For instance, a Java compiler scans a program to find identifiers, literals, and operators; and a spelling checker scans a document for errors. We will study a simple instance of this problem - scanning a string looking for upper case letters, which nonetheless will illustrate general techniques for scanning.

The problem requires us to decompose a string into characters and to process these characters. Let us first consider the type provided by Java to process characters.

char

The computer must often process characters. Most programs accept character input and produce character output. In fact, some programs, such as a spelling checker, perform most of their computations in terms of characters. Characters are the building blocks for the strings we have seen before.

This type defines a variety of characters including the English letters (both lowercase, a..z, and uppercase, A..Z), the decimal digits 0..9; “whitespace” characters such as blank and tab; separators such as comma, semicolon, and newline; and other characters on our keyboard. A character can be represented in a program by enclosing its print representation in single quotes:

```
'A' 'Z' '1' '0' ' '
```

Two consecutive single-quotes denote a special character called the *null* character:

```
''
```

The null character is used to mark the end of a string. It is not useful to print it since Java prints nothing for it.

How do we represent the single-quote character itself? We could enclose it in single-quotes:

```
'''
```

However, Java would match the first two single-quotes as the null character, and think you have an extra single-quote character. So, instead, it defines the following representation for a single-quote:

```
'\''
```

Here, instead of enclosing one character within quotes, we have enclosed a two-character escape *sequence*. The first of these characters, `\`, or *backslash*, is an *escape* character here, telling Java to escape from its normal rules and process the next character in a special way.

Java defines escape sequences to represent either those characters that cannot use the normal representation or those for which the normal representation may not be readable. A literal cannot have a new line character in it, so `\n` denotes the new line character. A backslash after the first quote denotes special processing, not the backslash character itself, so `\\` denotes the backslash character. Typing a backspace after a single-quote would erase the single-quote, so `\b` denotes the backspace character. We can represent the tab character by entering a tab between quotes:

`"\t"`

but this representation can be mistaken as the space character. So `\t` denotes a tab character. Similarly, we can represent the double quote character as:

`"\""`

but it may be mistaken for two null characters. So `\"` denotes the double quote character. The following table summarizes our discussion:

Escape Sequence	Character Denoted
<code>\'</code>	'
<code>\n</code>	new line
<code>\b</code>	back space
<code>\\</code>	\
<code>\t</code>	tab
<code>\"</code>	"

Table 1 Some Useful Java Escape Sequences

Java allocates 16 bits for storing a character. As a result, it can support as many as 2^{16} different characters, which is useful since we would like to represent characters of all current languages; and some of them such as Chinese have a large character set. It stores a non-negative integer code for each character. As programmers, we do not have to concern ourselves with the exact integer assigned to each character. However, as discussed later, we need to know something about the relative order in certain subsets of the character set such as the set of lower case letters and the set of digits.

Ordering Characters

Like numbers, characters are ordered. Clearly, it makes sense to order Java values that are numbers, but why order characters? In ordinary life, we do order characters, when we learn the alphabet, and more important, when we search directories. It is to support such searches that programming languages order the elements in the character set. The integer code, or *ordinal number*, assigned to a character is its position in this set. We do not need to know the exact ordinal number assigned to a character. It is sufficient to know that:

- The null character, `'\0'`, is assigned the ordinal number 0.
- The digits are in order.
- The uppercase letters, `'A' .. 'Z'`, are in order.
- The lowercase letters, `'a' .. 'z'`, are in order.
- Letters of other alphabets are in order.

Thus, we know that:

<code>'1' > '0'</code>	\rightarrow	true
<code>'B' > 'A'</code>	\rightarrow	true

`'a' > 'b'` \rightarrow `false`
`c >= ''` \rightarrow `true`

where `c` is a character variable holding an arbitrary character value.

Based on the information above, we cannot compare elements from different ordered lists. Thus, we cannot say whether:

`'A' > 'a'`
`'A' > '0'`

Converting between Characters and their Ordinal Numbers

Like other programming languages, Java lets you find out the exact ordinal number of a character by casting it as an `int`. Thus:

```
System.out.println ( (int) 'B' )
```

will print out the ordinal number of `'B'`. This cast is always safe, since `char` (16 unsigned bits) is a narrower type than `int` (32 bits). Therefore, when context demands `ints`, Java automatically performs the cast. Thus:

```
int i = 'A';
```

and:

```
'B' - 'A'
```

computes the difference between the integer codes of the two characters, returning 1. Java lets you directly perform all the `int` arithmetic operations on characters, and uses their ordinal numbers in the operations. Usually we do not look at absolute values or sums of ordinal number - the differences are more important, as we see below.

You can also convert an ordinal number to a character:

```
char c = (char) intCodeOfB;
```

We had to perform a cast because not all integers are ordinal numbers of characters, just as not all doubles are integers. For instance, the following assignment makes no sense:

```
char c = (char) -1
```

since ordinal numbers are non-negative values. Java simply truncates the 32 bit signed value into a 16 unsigned value, much as it truncates a `double` with a fraction part to an `int` without a fraction. Again, by explicitly casting the value you are telling Java that you know what you are doing and are accepting the consequences of the truncation.

The two-way conversion between characters and ordinal numbers can be very useful. For instance, we can find the predecessor or successor of characters:

```
(char) ('I' - 1) == 'H'  
(char) ('I' + 1) == 'J'
```

We can also convert between uppercase and lower case characters, which is useful, for instance, when processing languages in which case does not matter:

```
(char) ('i' - 'a' + 'A') == 'I'  
(char) ('I' - 'A' + 'a') == 'i'
```

To understand why the above equalities hold, consider an equality we know is true based on the fact that letters are ordered:

```
'i' - 'a' == 'I' - 'A'
```

Moving the 'a' to the right, we get:

`'i' == 'I' - 'A' + 'a'`

Moving the 'A' to the left we get:

`'i' - 'a' + 'A' == 'I'`

Indexing

This problem requires a way to decompose a string into its constituent characters. We identify string components through their positions or *indices* in the string. For `Strings`, the following method is provided, takes the component index as an argument and returns a character:

public char charAt (int);

Not that string indexing is different from array indexing.

String indices start from 0 rather than 1. Thus, if:

`String s = "hello world"`

`s.charAt(0) == 'h';`

`s.charAt(1) == 'e'`

In general, we access the *i*th character of string, *s*, as:

`s.charAt(i-1);`

Not all string indices are legal. An index that is smaller (greater) than the index of its first (last) character is illegal. Thus, both of the following accesses will raise a `StringIndexOutOfBoundsException` :

`s.charAt(11)`

`s.charAt(-1)`

The instance function, `length()`, returns the number of characters in a string. Thus:

`"helloworld".length() == 11`

`"".length() == 0`

We can use this function to define the range of legal indices of an arbitrary string *s*:

`0 .. (s.length() - 1)`

Sub-Strings

Besides individual characters, we may also wish to retrieve *sub-strings* of a string, that is, sequences of consecutive characters that appear in the string. The Java function:

public String substring (int beginIndex, int endIndex)

when invoked on a string, *s*, returns a new string that consists of the character sequence starting at `beginIndex` and ending at `endIndex - 1`, that is:

`s.charAt(beginIndex) .. s.charAt(endIndex - 1)`

It raises the `StringIndexOutOfBoundsException` exception if `beginIndex` is greater than `endIndex`. If they are both equal, it returns the empty string. Thus:

`"hello world".substring(4, 7) → "o w"`

`"hello world".substring(4, 4) → ""`

`"hello world".substring(7, 4)` throws `StringIndexOutOfBoundsException`

While `String` provides getter methods to read string characters and sub strings, it provides no setter method. This is because Java strings are readonly or *immutable*, that is, they cannot change. A separate class, `StringBuffer`, which we will not study in this course, defines mutable strings. The class `String` does, as we have seen before, provide the `+` operation on strings to create *new* strings from existing strings. Thus:

`"hello" + "world" == "hello world"`

Here, we do not change either string, but instead, create a new string that stores the result of appending the second string to the first one.

For Loop

The following program fragment illustrates string manipulation, printing the characters of a string in separate lines:

```
int i = 0; // initialization of loop variables
while (i < s.length()) { // continuation condition
    System.out.println (s.charAt(i)); // real body
    i++; // resetting loop variables
}
```

This is an example of a counter-controlled loop, since the variable `i` serves as a counter that is incremented in each iteration of the loop. It is also an example of an *index-controlled loop*, which is a special case of a counter-controlled loop in which the counter is an index .

Let us look at this code in more depth to grasp the general nature of a loop. We can decompose it into the following components:

1. *Continuation condition*: the boolean expression that determines if the next iteration of the loop should be executed.
2. *Initialization of loop variables*: the first assignment of variables that appear in the continuation condition and are changed in the loop body.
3. *Resetting loop variables*: preparing for the next iteration by reassigning one or more loop variables
4. *Real body*: the rest of the while body, which does the "real-work" of each iteration.

The while loop separates the continuing condition from the rest of the components, thereby ensuring we do not forget to enter it. However, it does not separate the remaining components - we have added comments to do so in the example above. As a result, we may forget to create one or more of these components. For instance, we may forget to reset the loop variables, thereby creating an infinite loop.

Java provides another loop, the *for loop* or *for statement*, illustrated below, which explicitly separates all four components:

```
for (int i = 1; i < s.length(); i++) {
    System.out.println (s.charAt(i));
}
```

This loop is equivalent to the program fragment above. In general, a for statement is of the form:

```
for (S1;E;S2) S3
```

It is equivalent to:

```
S1;
while (E) {
    S3;
    S2;
}
```

In other words, the for loop first executes S1 and then essentially executes a while loop whose condition is E and whose body is S3 followed by S2. S1 and S2 are expected to initialize and reset, respectively, the loop variables.

In comparison to a while loop, a for loop is more compact. More important, its structure reminds us to enter S1 and S2, that is, initialize and reset the loop variables.

Any of the three parts of a for loop, S1, E, and S2 can, in fact, be missing. A missing S1 or S2 is considered a null statement, while a missing E is considered true. Thus:

```
for (;) {
    S3;
```

```

    }
is equivalent to:
    while (true) {
        S3;
    }

```

The break statement can be used to exit both kinds of loops.

Scanning

Let us return to the scanning problem given above.

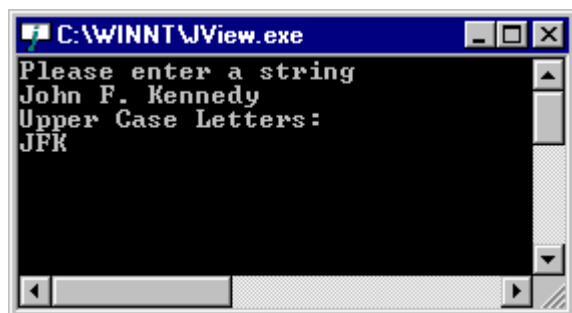


Figure 9 Finding Upper Case Letters

This is a simple example of a problem that involves scanning an *input stream* for *tokens*. A stream is a sequence of values of some type, T; and a token in a sequence of consecutive stream elements in which we are interested. The sequence of tokens generated from the input stream forms another stream called the *token stream*. Thus, the process of scanning involves converting an input stream to a token stream.

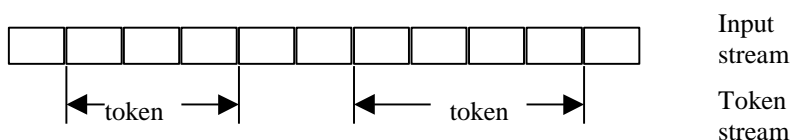


Figure 10 The Concept of Scanning

In the problem we have to solve, the input stream is the sequence of characters in the argument string, and the tokens in which we are interested are upper case letters. The tokens are one-character long since each uppercase letter forms a separate token, as shown in Figure 4. Our job, in this chapter, will be to write a scanner for it.

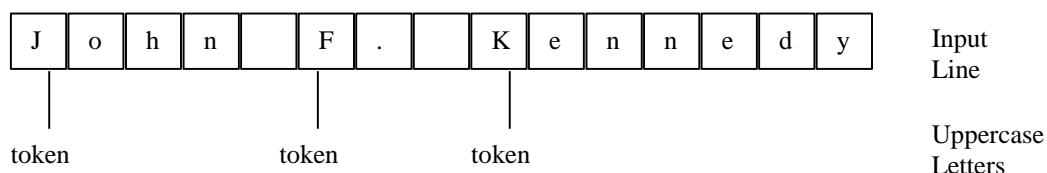


Figure 11 Scanning a Line of Characters for Upper Case letters

Scanning problems occur in many familiar domains. As we have seen, interactive programs scan the input for lines, integers, and other kinds of values. Similarly, file-based programs must often scan a file for various values. Word processors scan documents for occurrences of a particular string. Perhaps the most important application is in compilers, which scan programs for identifiers, literals, operators, and other tokens.

We are now ready to look at the scanner code:

```
package warmup;
public class AnUpperCasePrinter {
    public static void main(String[] args){
        if (args.length != 1) {
            System.out.println("Illegal number of arguments:" + args.length + ". Terminating program.");
            System.exit(-1);
        }
        System.out.println("Upper Case Letters:");
        int index = 0;
        while (index < args[0].length()) {
            if (isUpperCase(args[0].charAt(index)))
                System.out.print(args[0].charAt(index));
            index++;
        }
        System.out.println();
    }
    public static boolean isUpperCase(char c) {
        return (c >= 'A') && (c <= 'Z');
    }
}
```

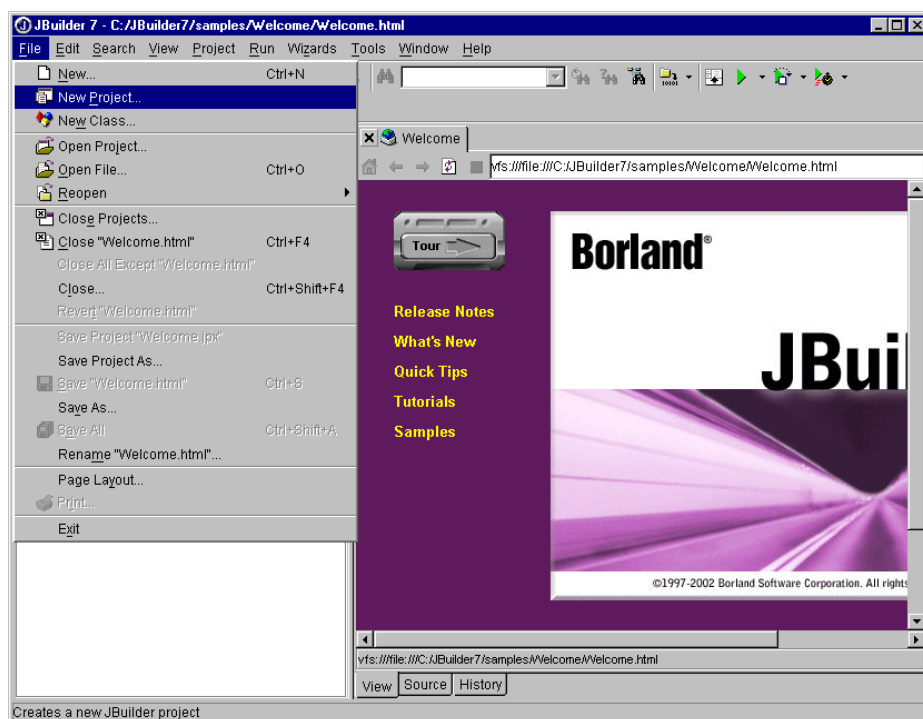
The program uses a separate method, `isUpperCase()`, to check if a character is an upper case letter. As in the main method, we use the keyword **static** since we are not yet doing object-based programming. However, we do not use the keyword **void** this time. Instead, we use a type name. The reason is that this method is a function that returns a value. The type name in the header indicates that a boolean value is returned. The return statement in the method actually returns the value.

Using JBuilder

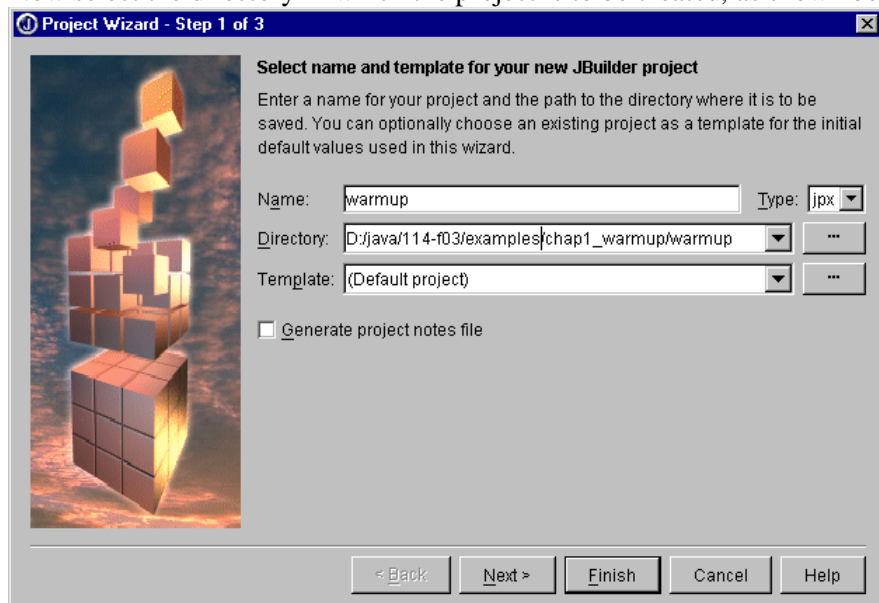
Getting Started

We saw earlier how to develop programs using the bare none environment. Let us see how this process occurs with the Borland JBuilder environment.

The first step is to execute the New Project command, as shown below.

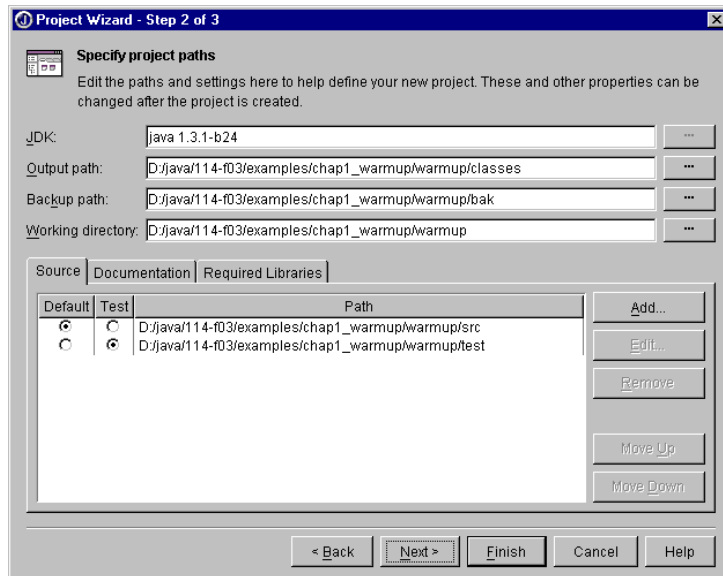


Now select the directory in which the project is to be created, as shown below.

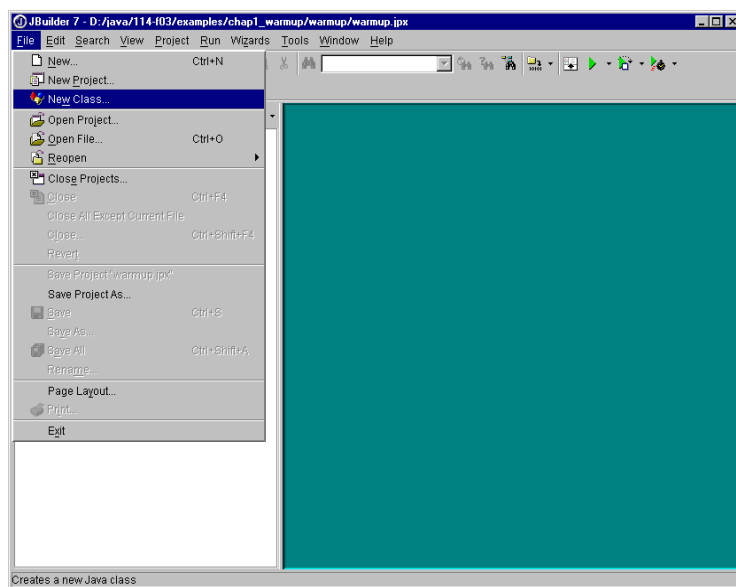


The component part of the directory name becomes the package name for the classes you create. In the example above, warmup becomes the name of the package. The directory associated with the package is automatically created by JBuilder.

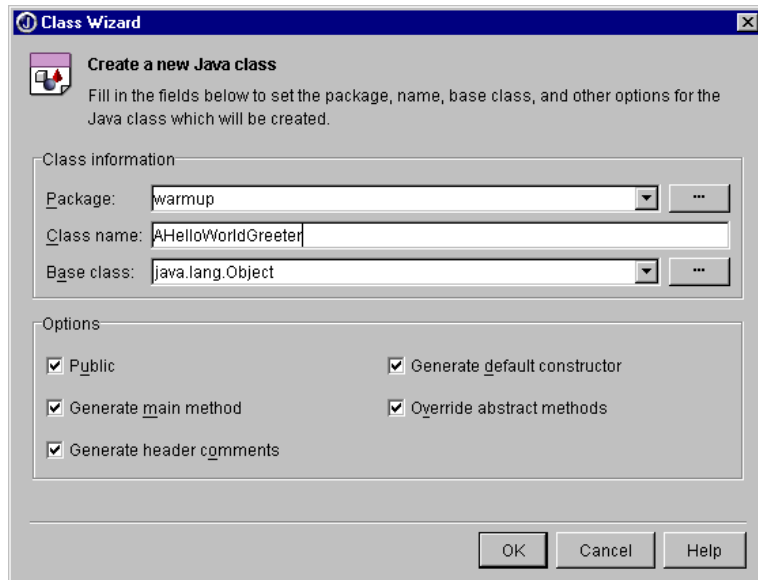
You don't need to worry about the next screen presented to you:



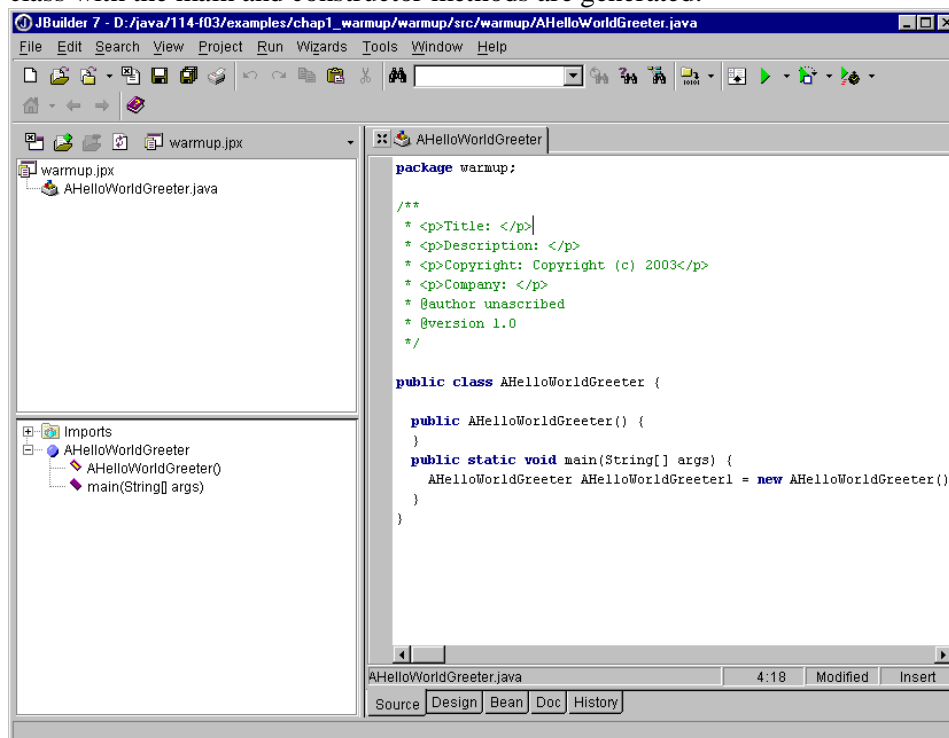
The finish button takes you to a newly created but empty project. Let us execute the New Class command:



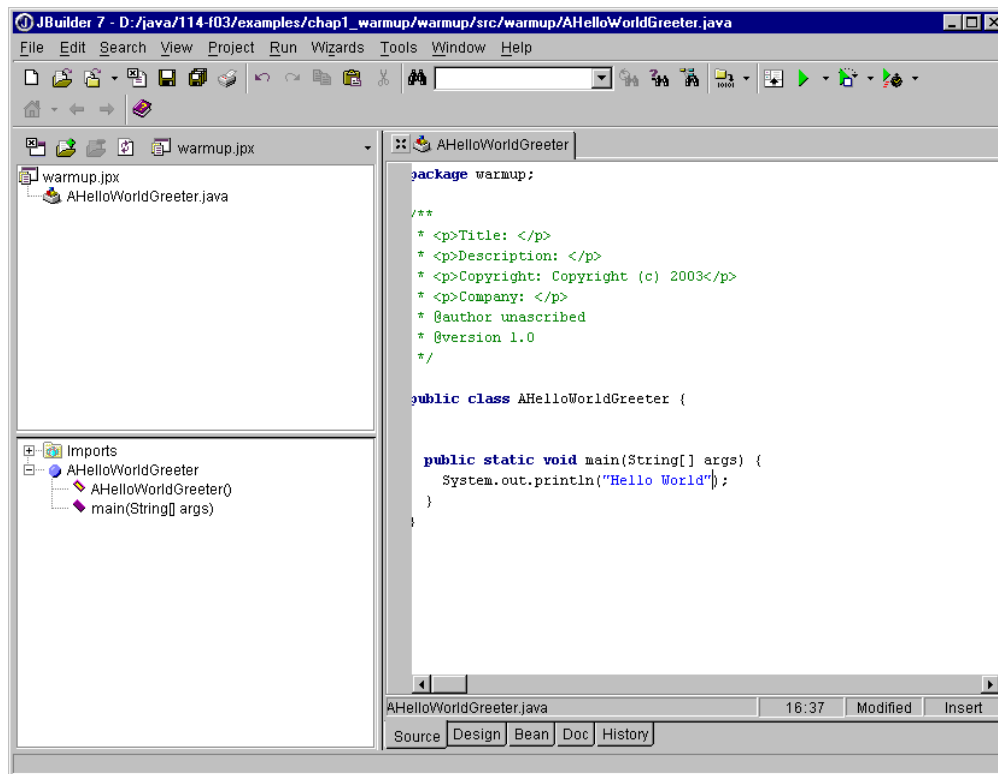
We now enter the name of the class and the package in which it is to be put:



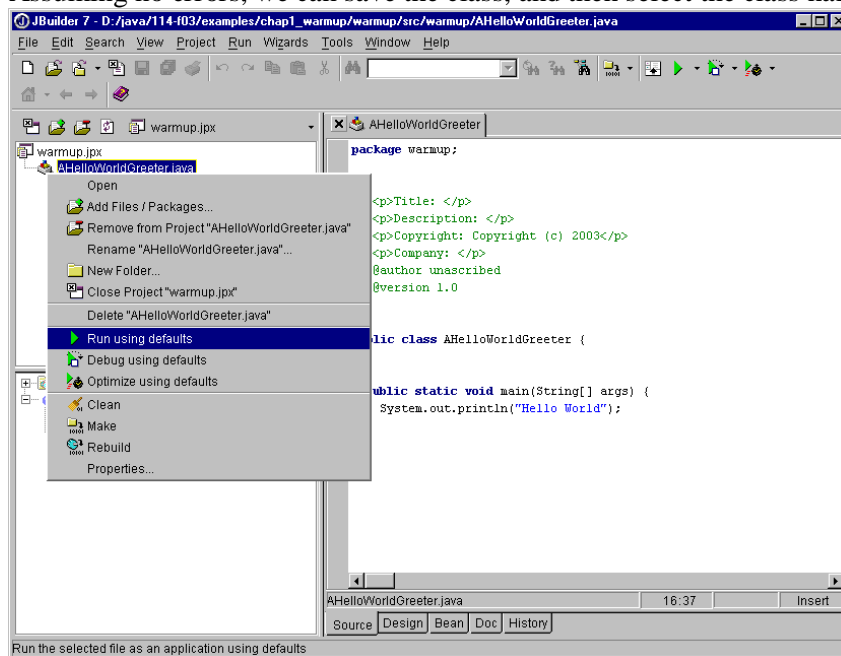
Because the generate main method and default constructor boxes are checked, when we press OK, a new class with the main and constructor methods are generated:



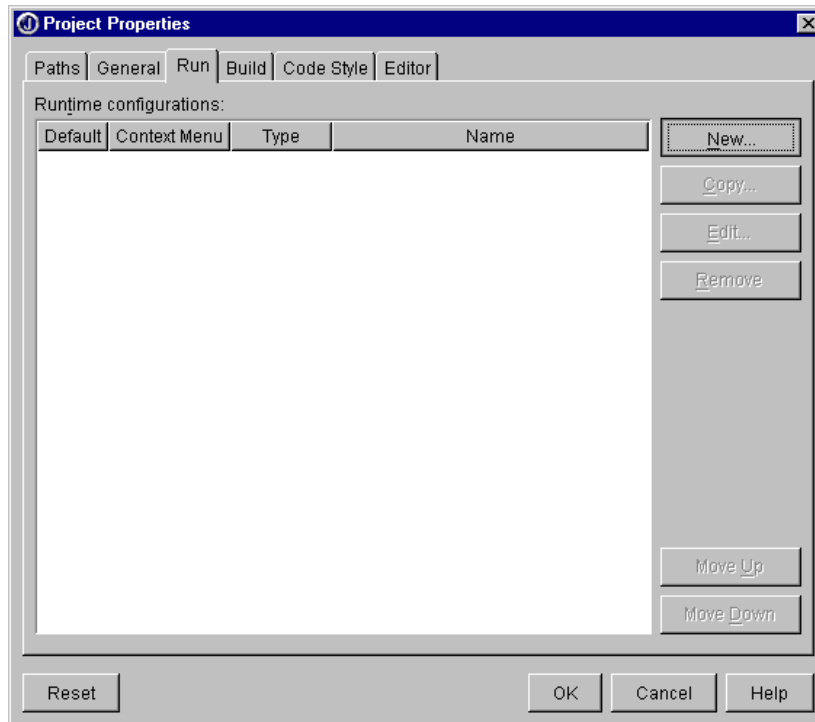
We can edit these contents:



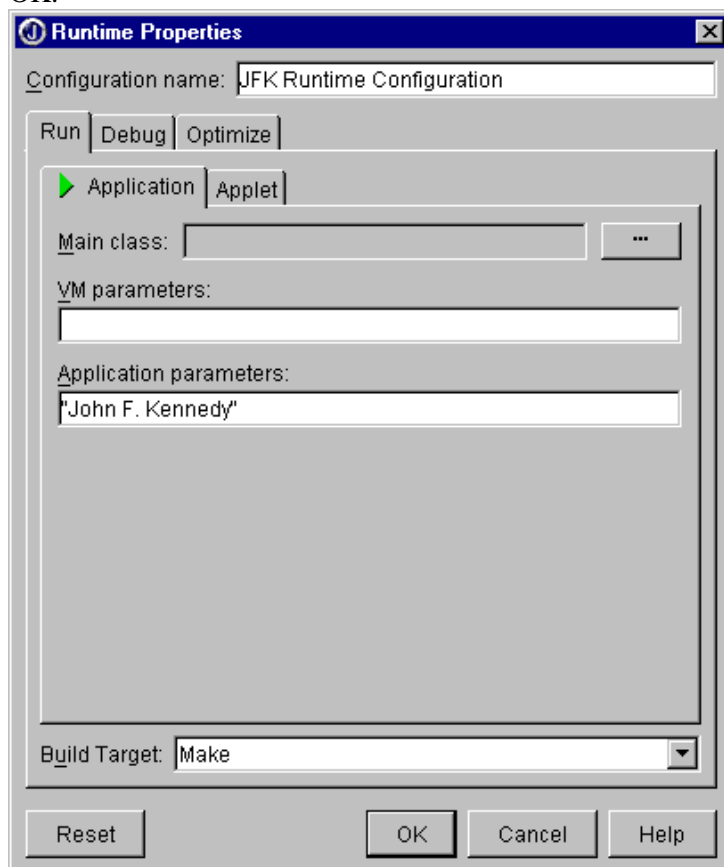
Assuming no errors, we can save the class, and then select the class name and run it:



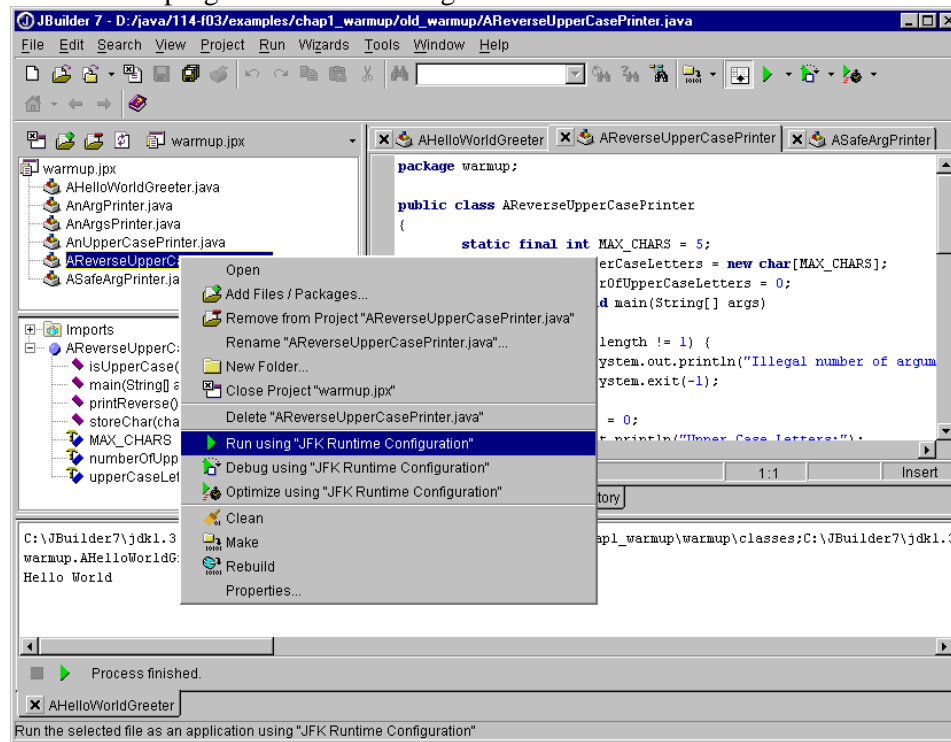
The result of the execution is shown below:



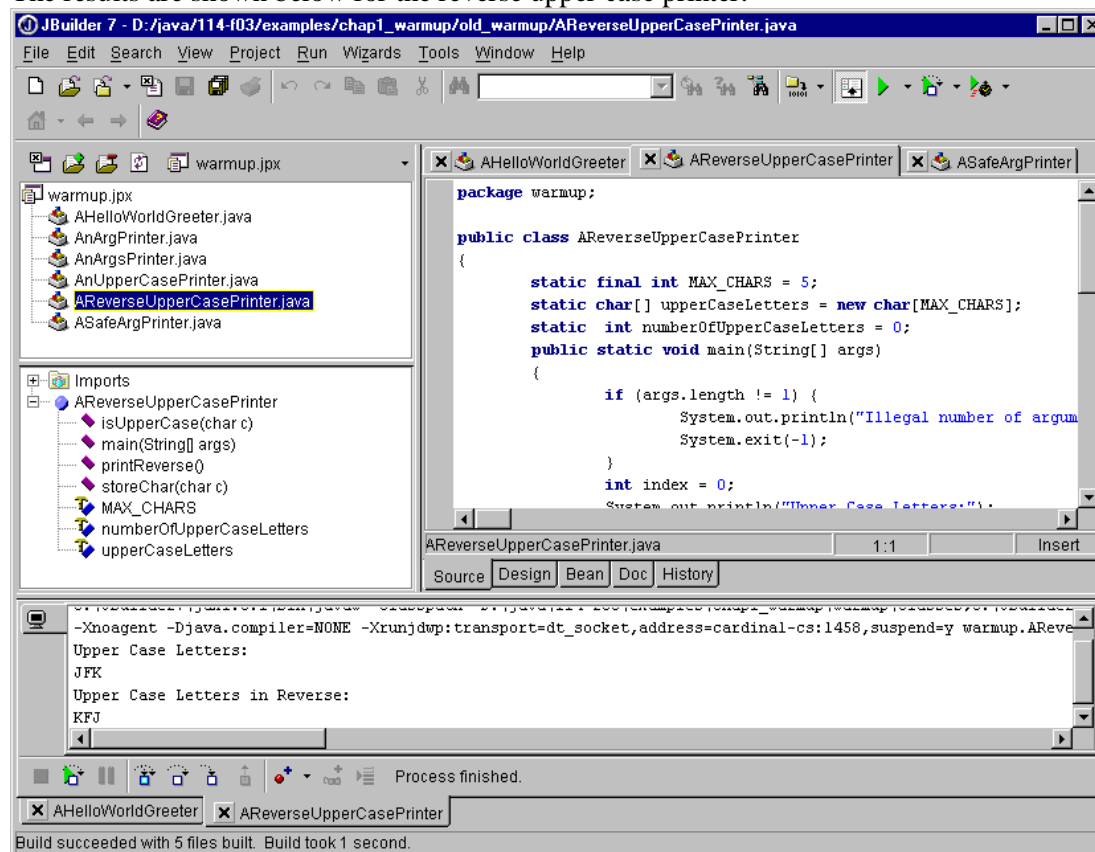
Enter the arguments under application parameters and enter configuration name, as shown below and hit OK.



Now run the program with this configuration:



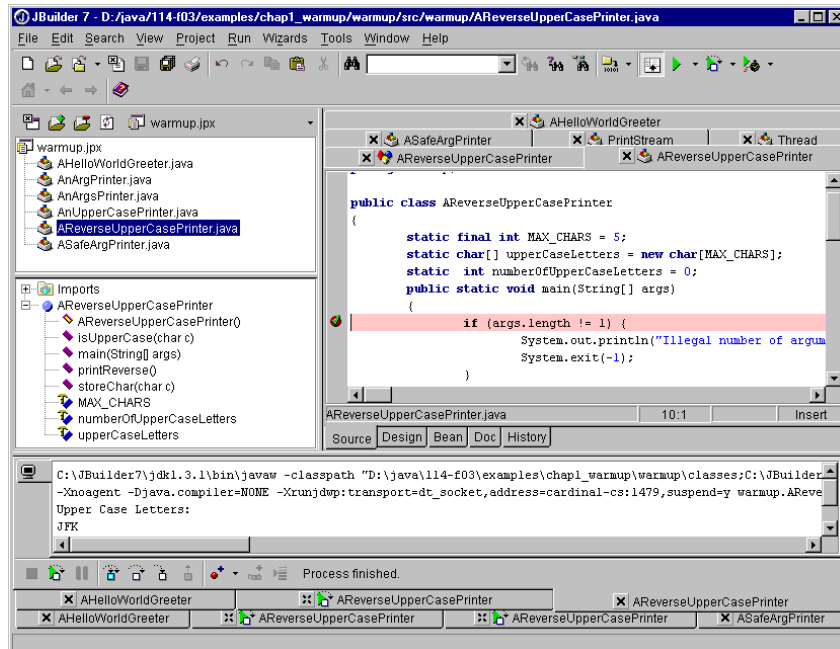
The results are shown below for the reverse upper case printer:



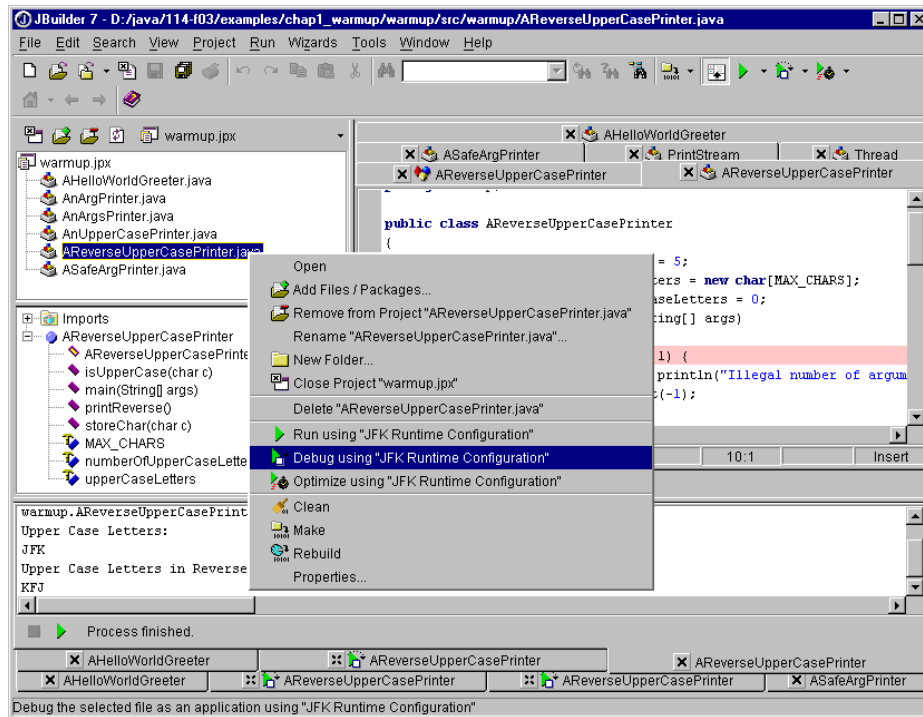
Using the Debugger

Seldom does one get the right results on the first try, especially for a program of the complexity you will develop in this class. Let us see next how to use a debugger to run a program.

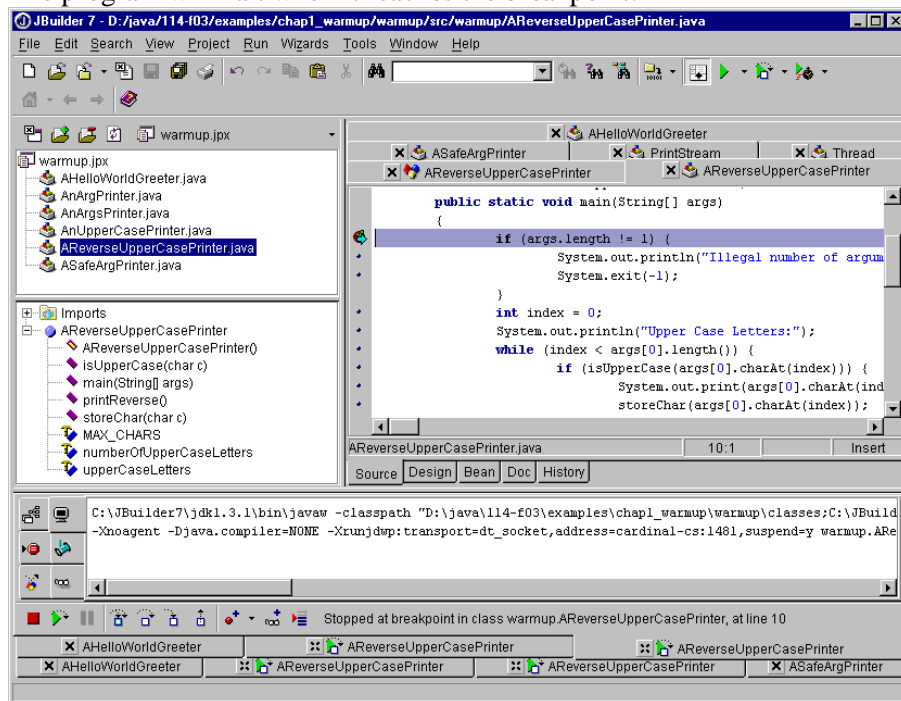
When you see an error, the first thing you might do is narrow the cause to the start of some statement sequence. You can then set a *breakpoint* at the start of that sequence. You can set a breakpoint by simply clicking next to it in the vertical breakpoint bar. An indication is then given on the bar where you clicked and the statement is highlighted:



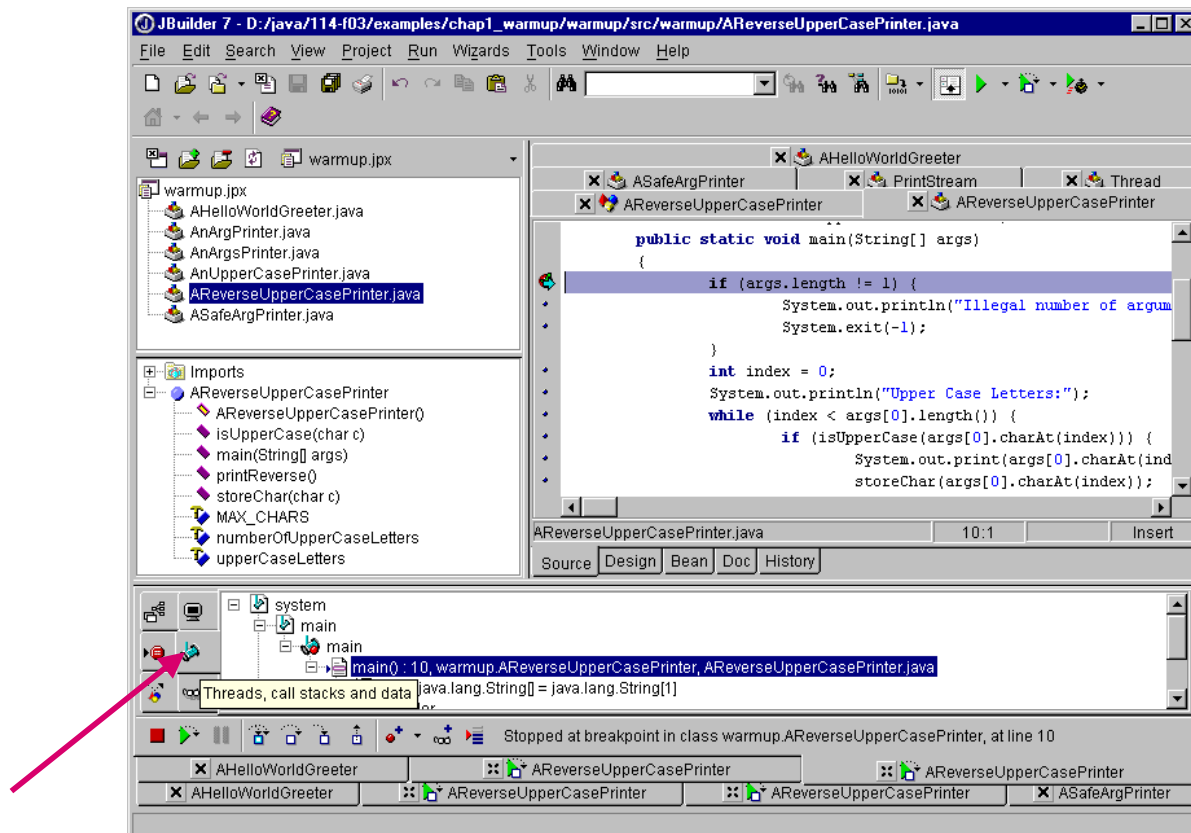
If you now run the program in the debugging mode:



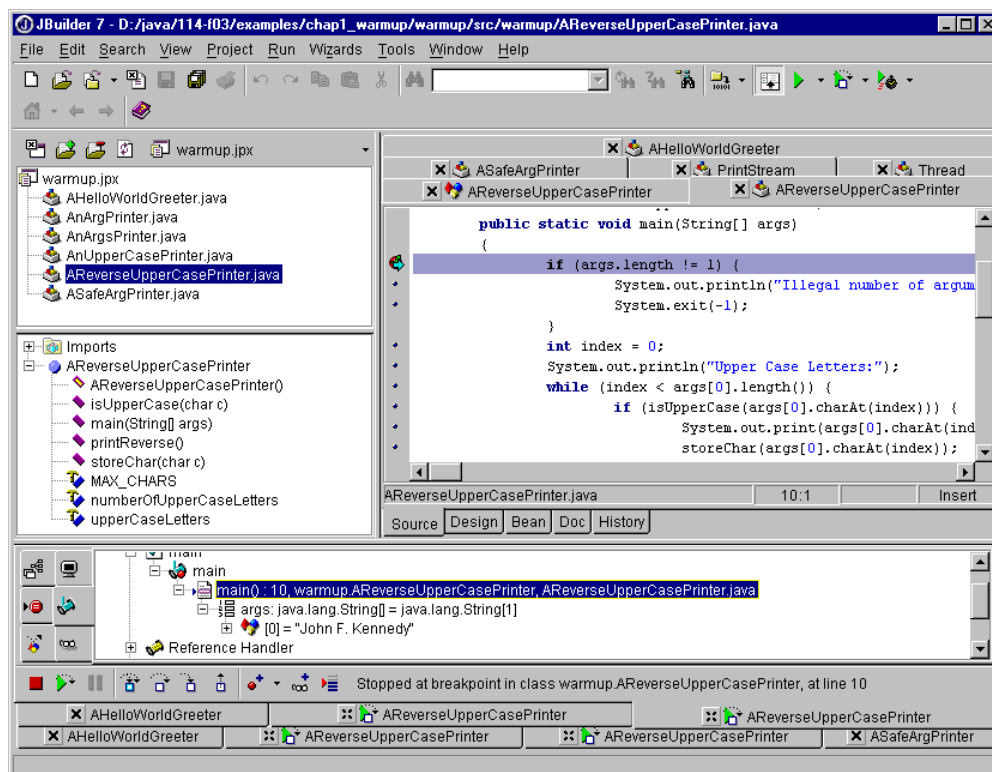
The program will halt when it reaches the breakpoint:



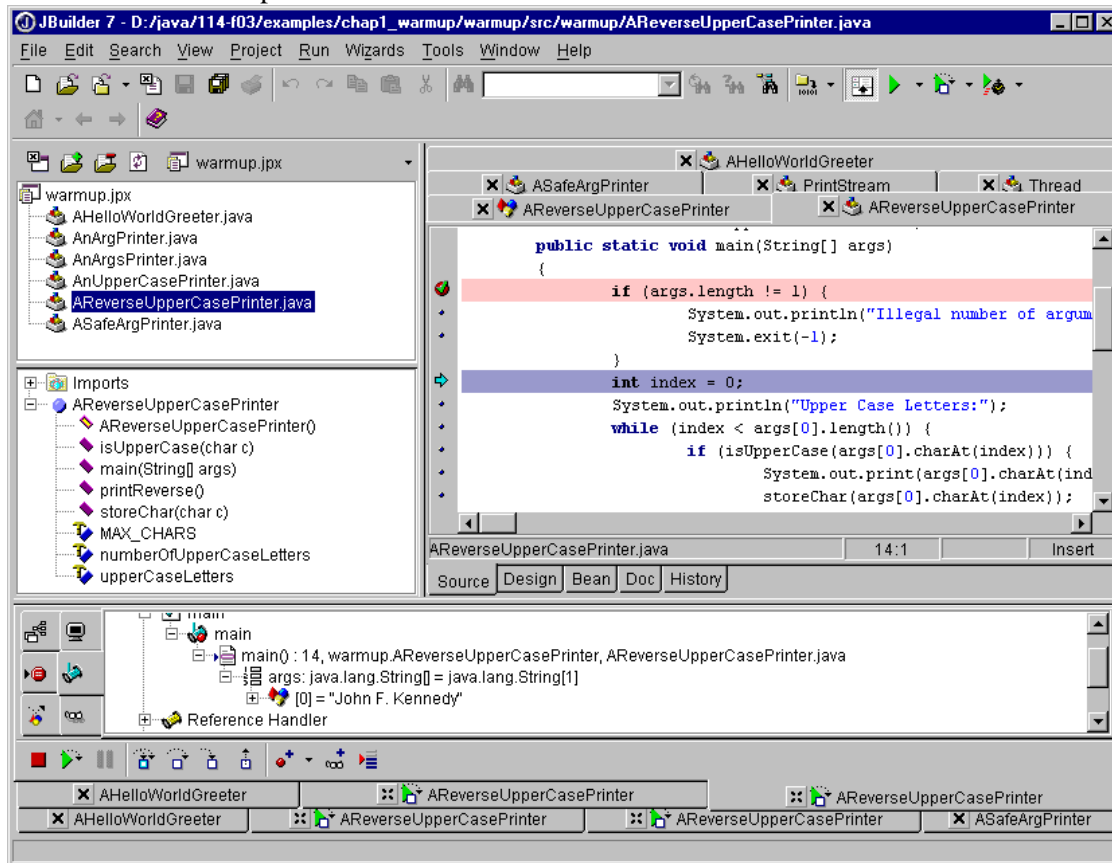
The green arrow next to the statement and its blue highlight indicate that it is about to execute it. You can examine the call stack by pressing the button indicated by the arrow below:



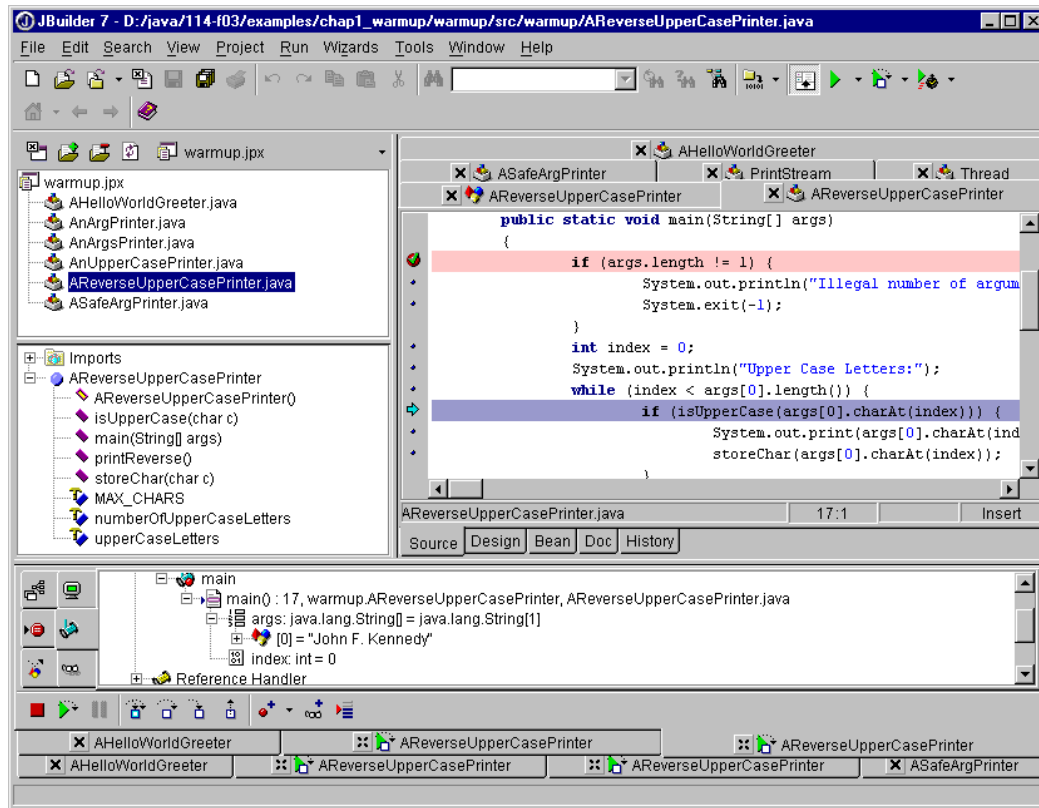
You can now navigate the call stack to look at the values of parameters:



For instance, the picture above shows that `args[0]` parameter of `main` is "John F. Kennedy". Once we are satisfied that the program state is as we expected, we can step to the next statement by pressing F8, the accelerator for the step over command:

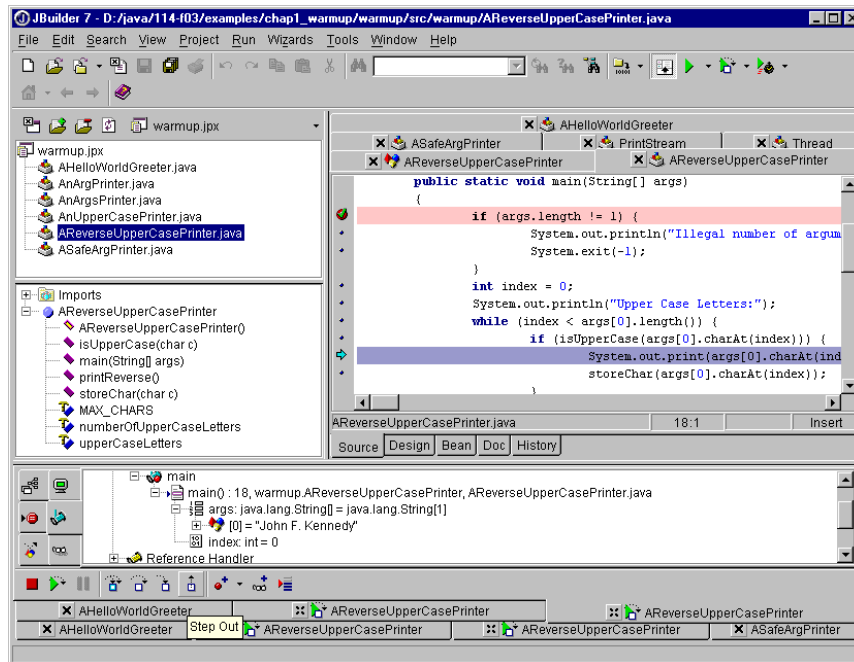


As we see in the figure, the blue highlight and the green arrow move to the next statement to be executed. Let us keep executing F8 until we reach the following statement:

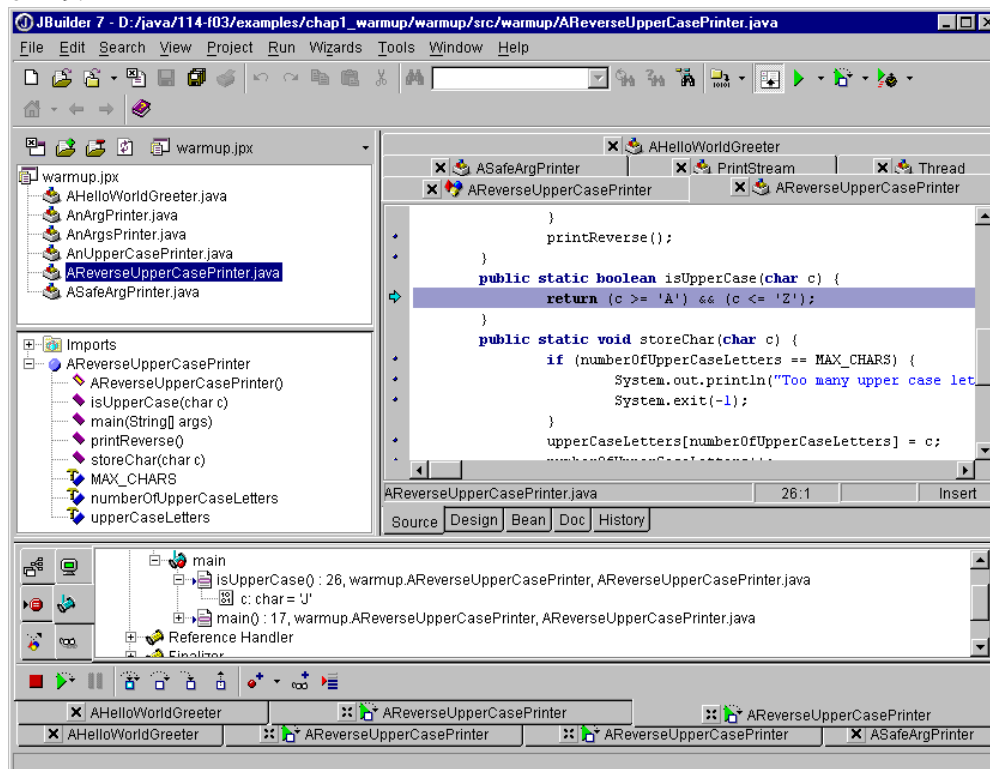


As shown in the call stack, the value of the index variable is 0 at this point, as this is the first time we are going through the loop. If we keep single stepping, we can see how many times the loop is executed.

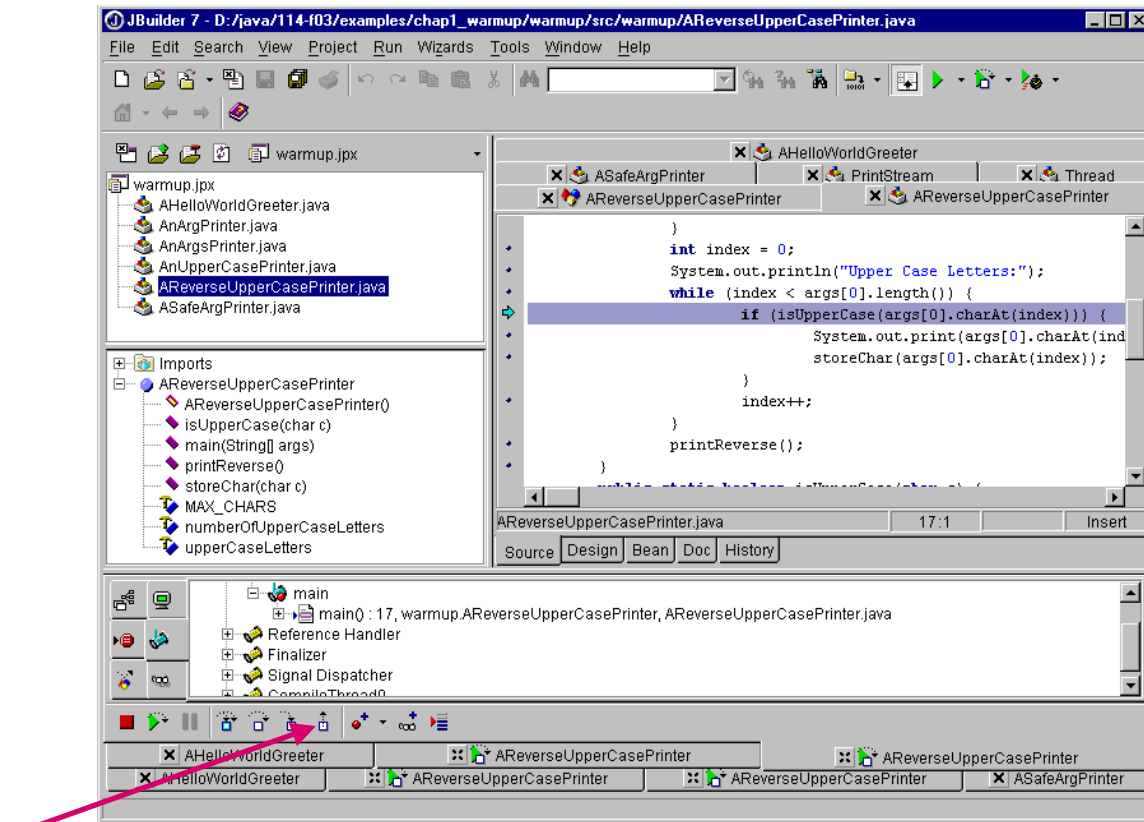
What should happen if we press F8 at this point to go to the next statement? Should it go to the next statement in the main method or first statement in the `isUpperCase()` method? As shown below, the answer is the former:



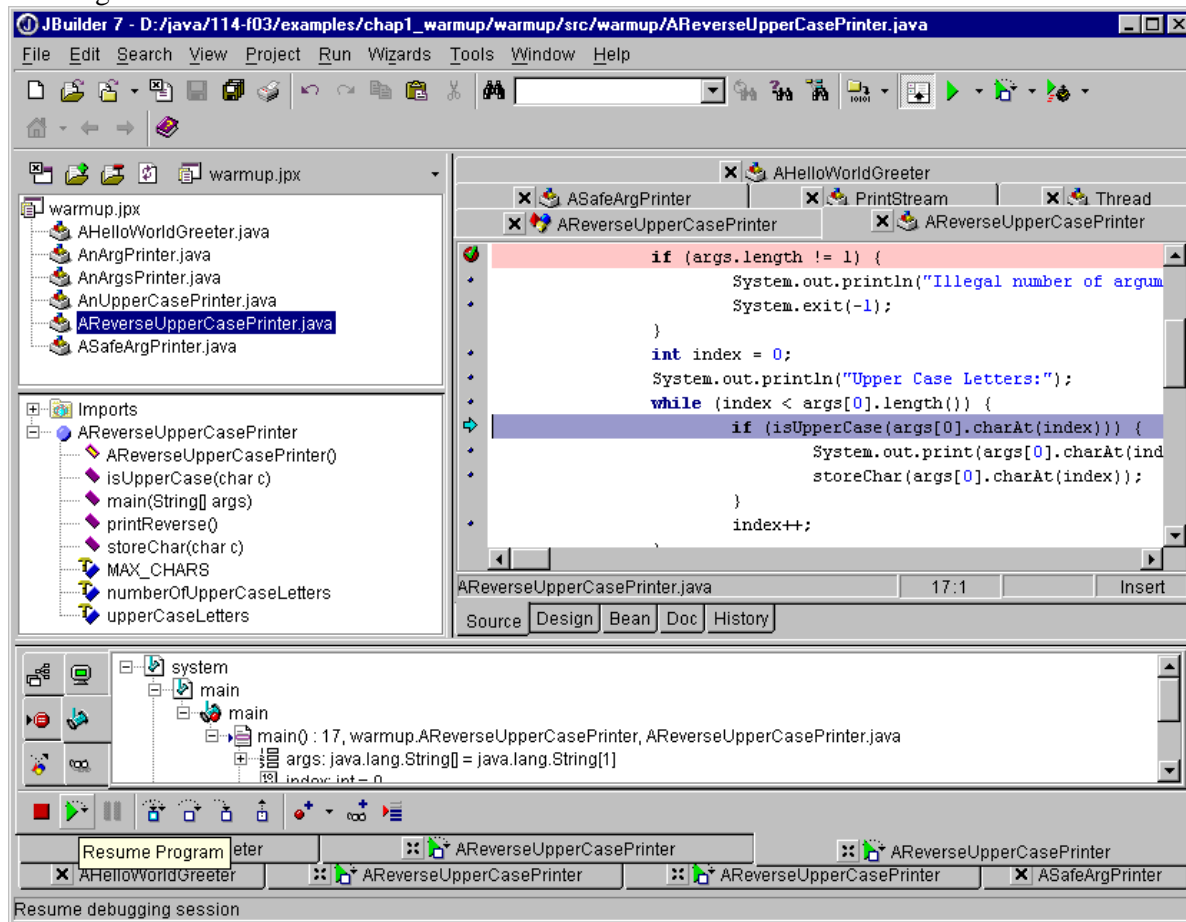
The reason is that F8 steps over any called method. If we want to step into the called method, we must click on F9:



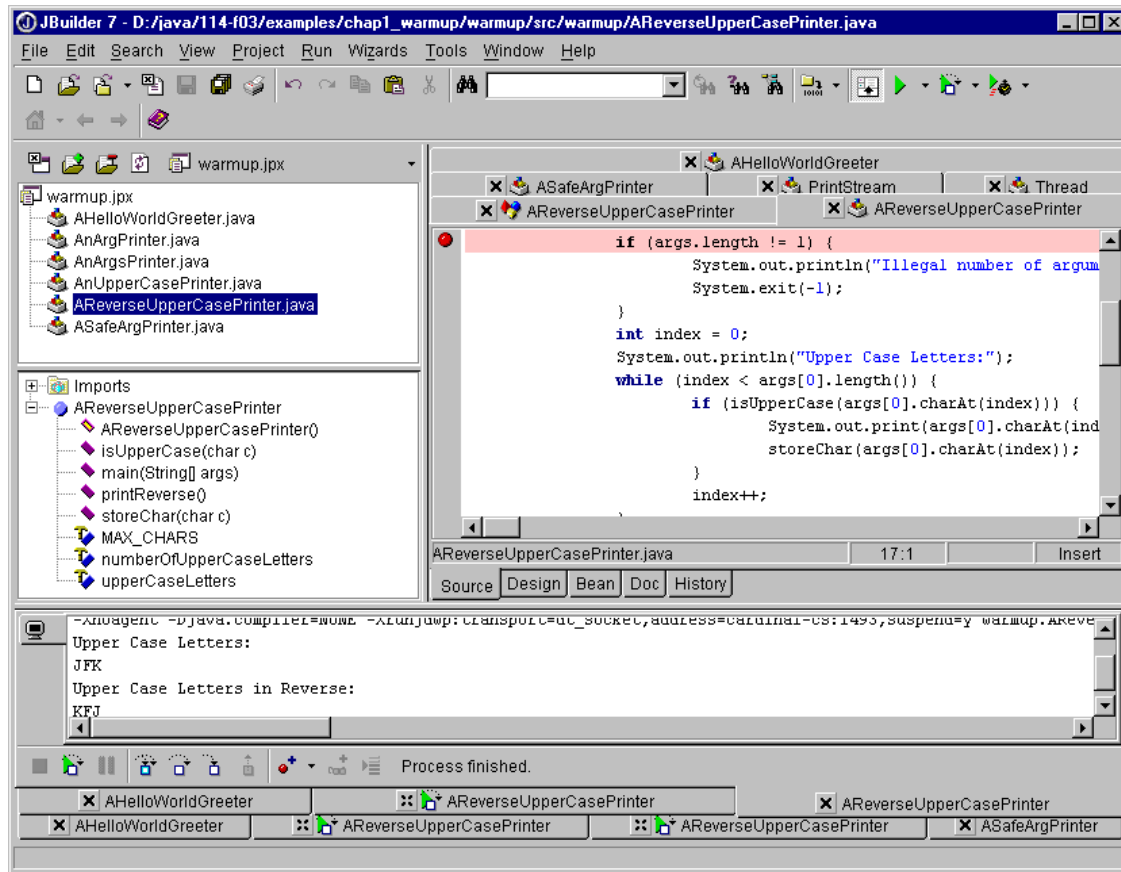
We can step out of the called method to the calling method by clicking on the icon shown indicated by the arrow below:



Clicking on the button indicated below



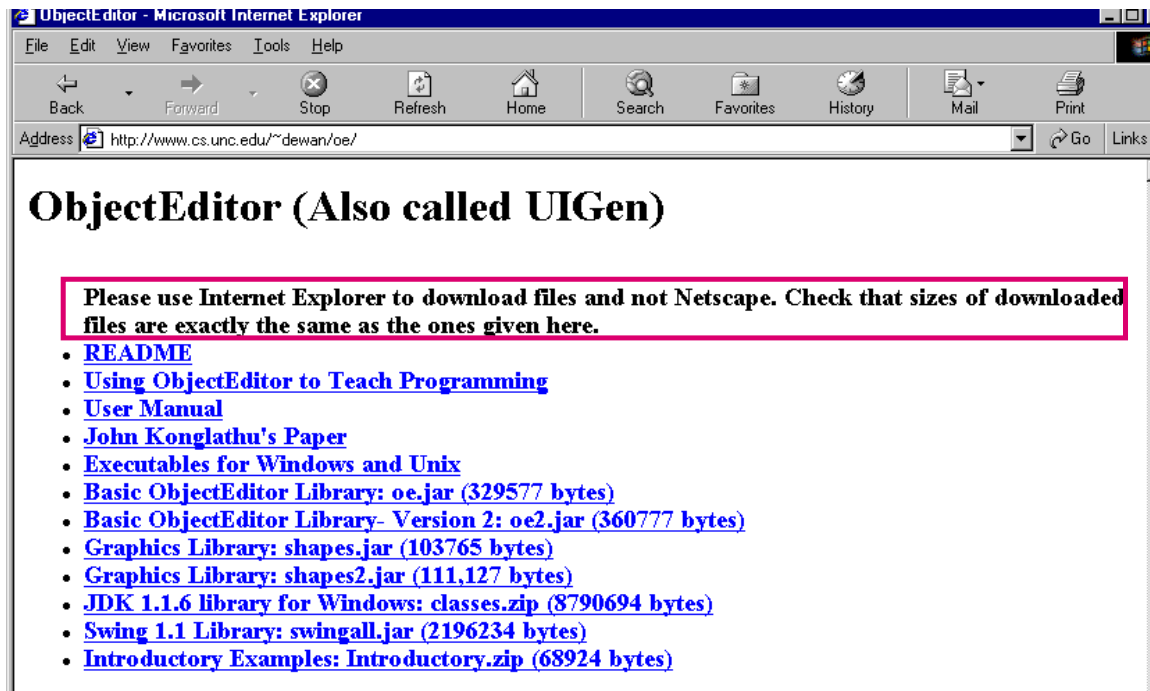
takes us out of the single step mode to the next breakpoint, if one exists, or the end of the program otherwise:



Installing a Library

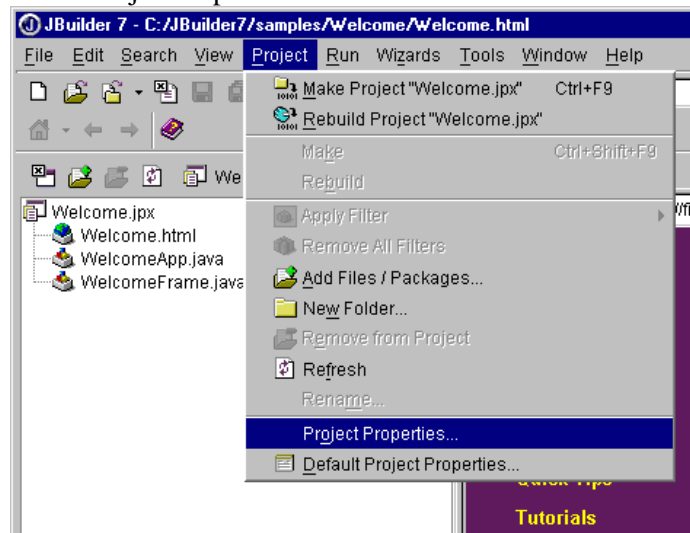
In this course, we will be using a system called ObjectEditor, which was developed by my research group. To use it, we must tell our programming environment to make it part of the library used in our project. The following screenshots illustrate this process for ObjectEditor.

Before you create a JBuilder library, you must load the code for it on your computer. In case of ObjectEditor, the code is stored in two files: `oe.jar` and `shapes.jar`, which can be obtained from: <http://www.cs.unc.edu/~dewan/oe>.

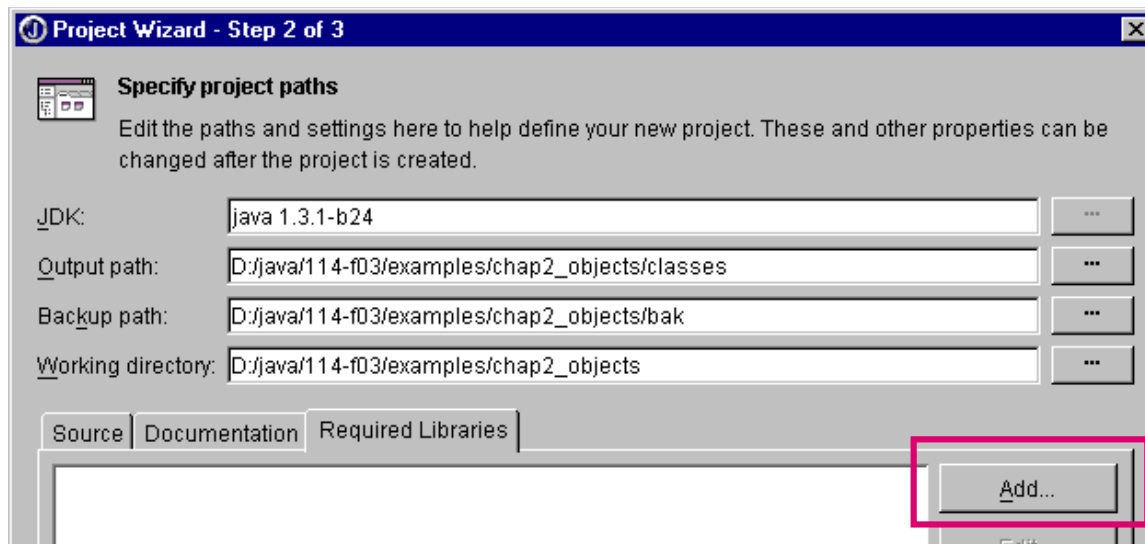


The file oe2.jar contains a newer version of the code, but may have more bugs. oe.jar should be sufficient for your assignments.

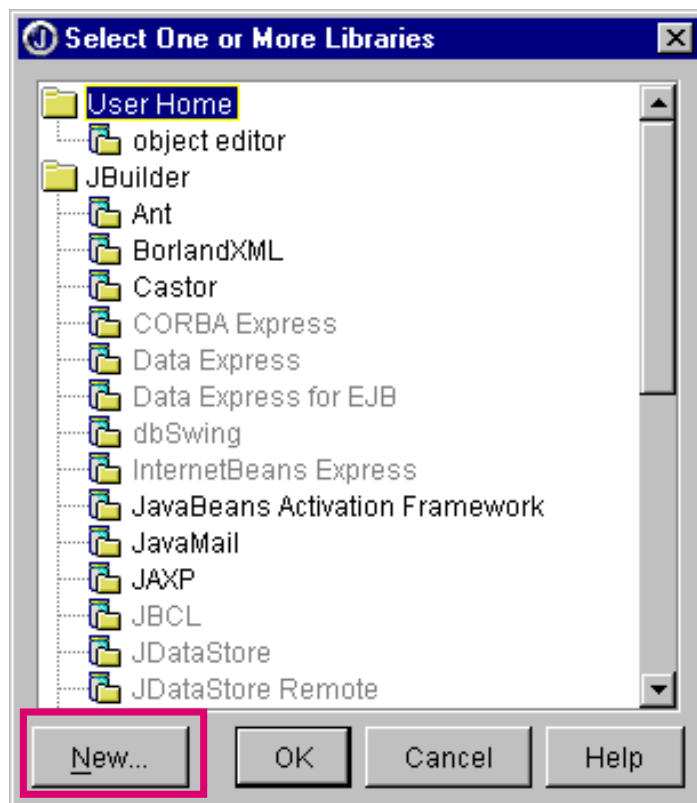
Select ProjectProperties.



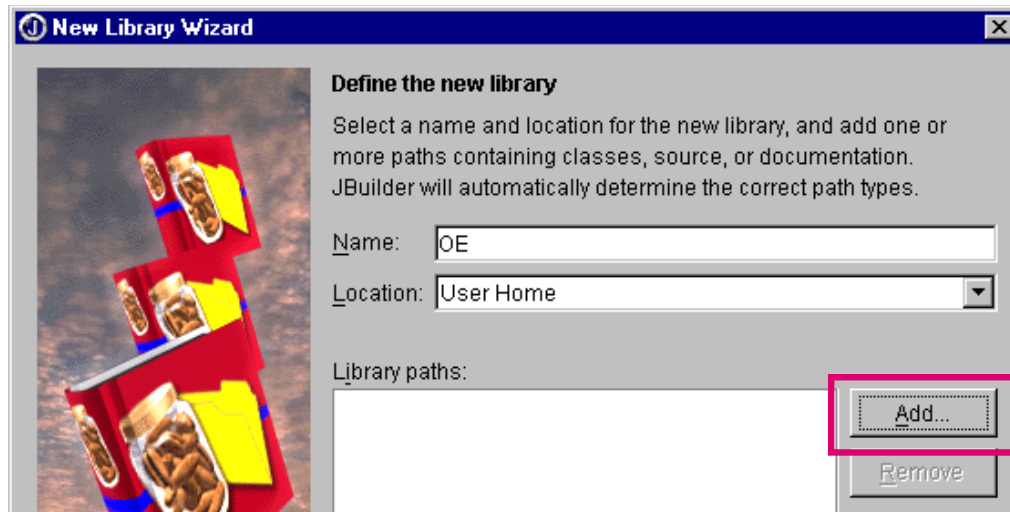
Press the Add button.



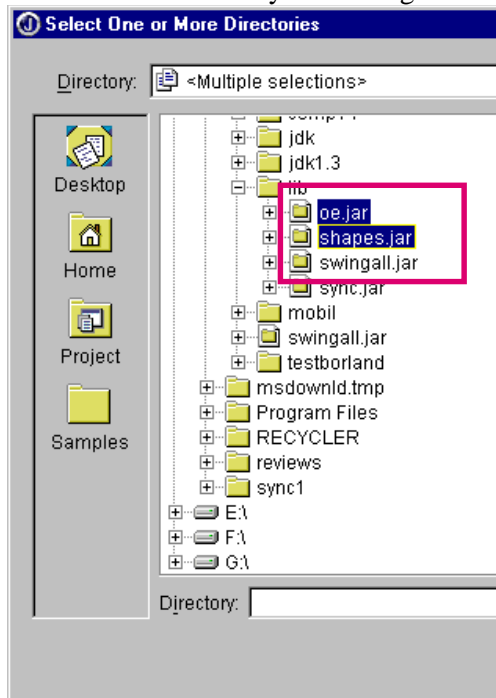
The new window shows the list libraries available for use. The ObjectEditor library was previously added by me under the name object editor. To illustrate the process of creating a new library, we will add it again, under a new name. Press the new button.



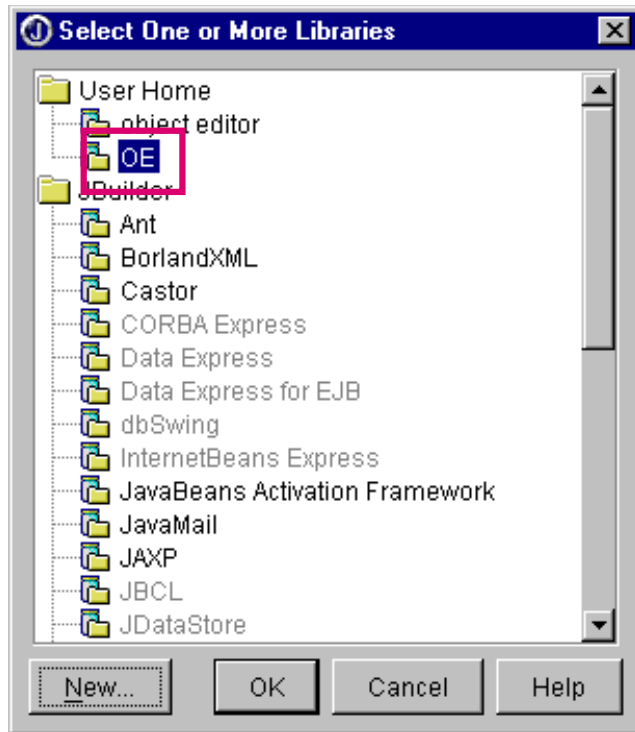
Give a name to the library and press the Add button.



Browse to the directory containing the library files.



The two files for ObjectEditor are oe.jar and shape.jar. Select these and press Ok. You see the new library added to the list of libraries. Select it, and press OK to add it to your project.:



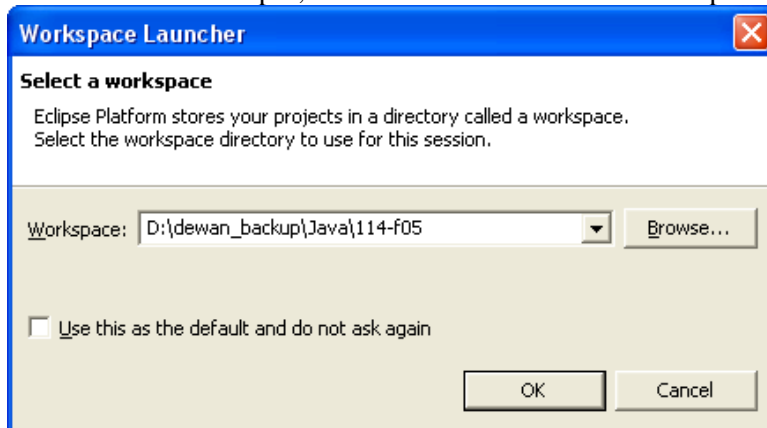
Now unwind back through the various windows you created.

Using Eclipse

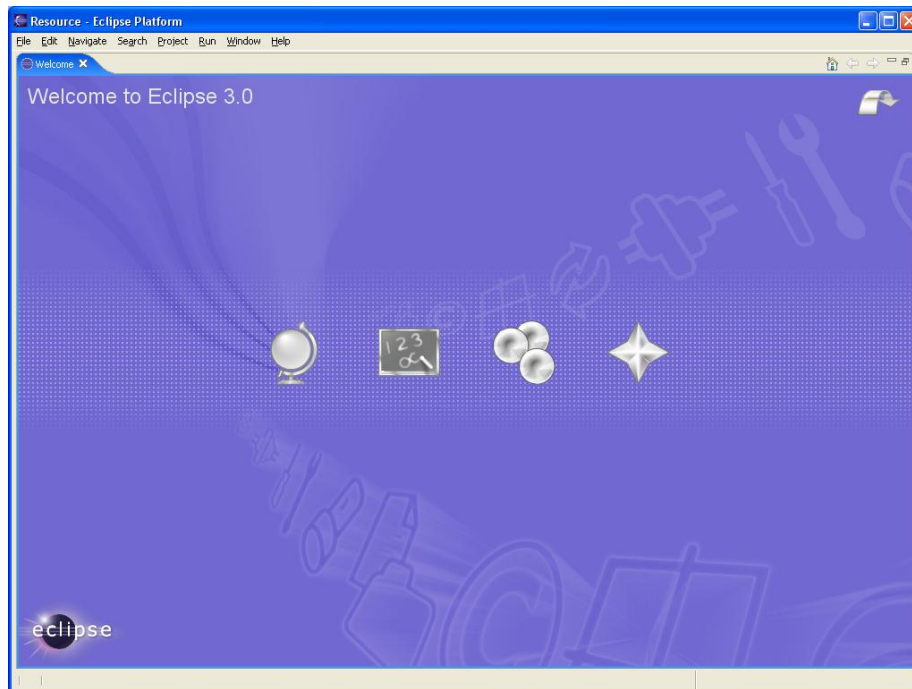
Getting Started

We saw earlier how to develop programs using the bare none environment. Let us now see how t occurs with the Eclipse environment.

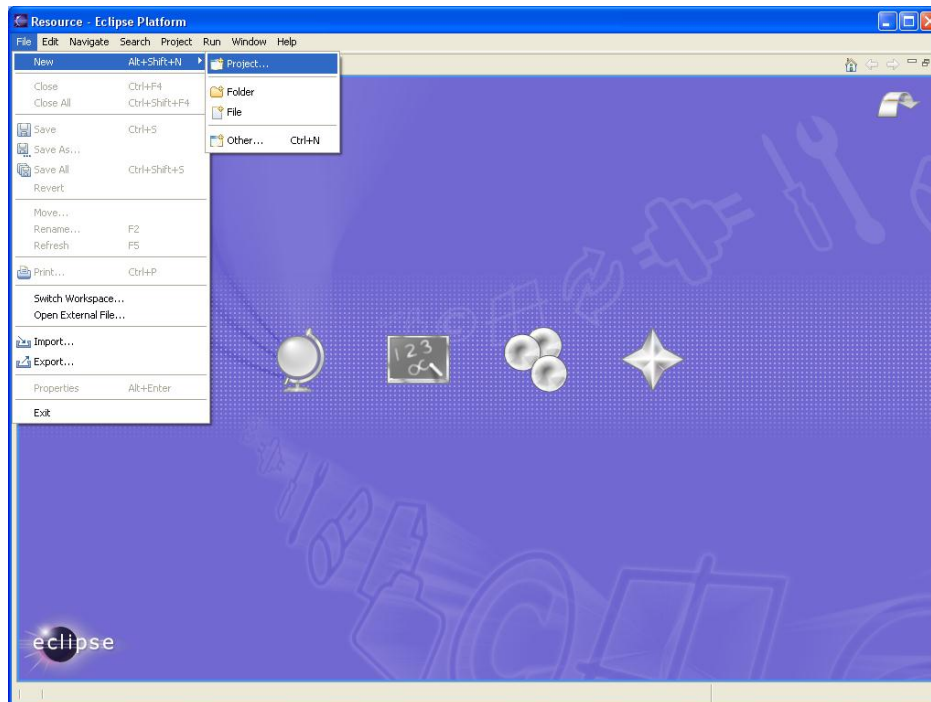
When you start Eclipse, it will ask you for the name of the workspace - the folder in which all of your projects will be stored. It will initialize the workspace to a default one or the last one you worked on. As this is out first use of Eclipse, let us enter the name of the workspace folder:



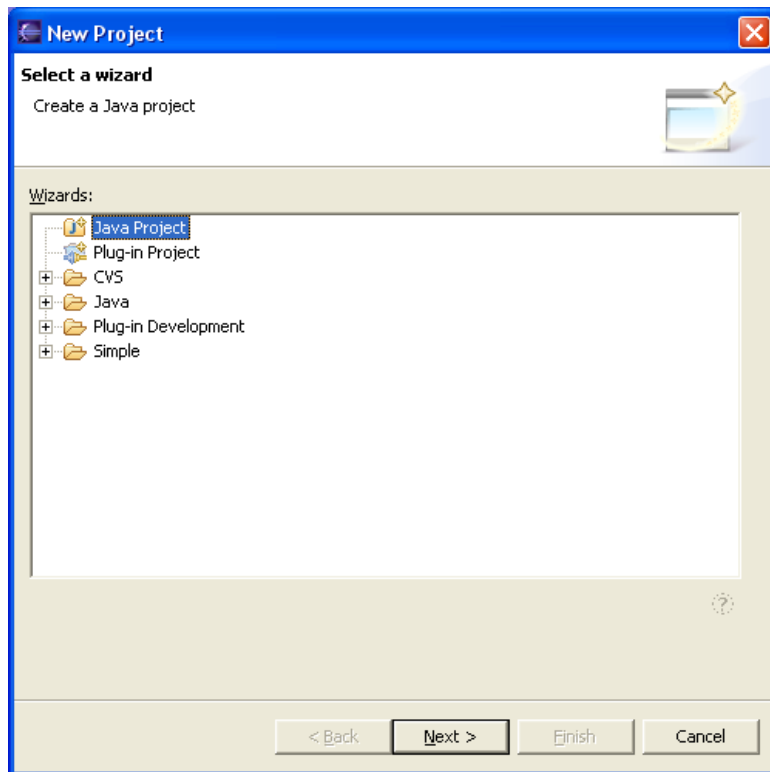
As our workspace currently contains no projects, the initial screen shows us only a welcome message:



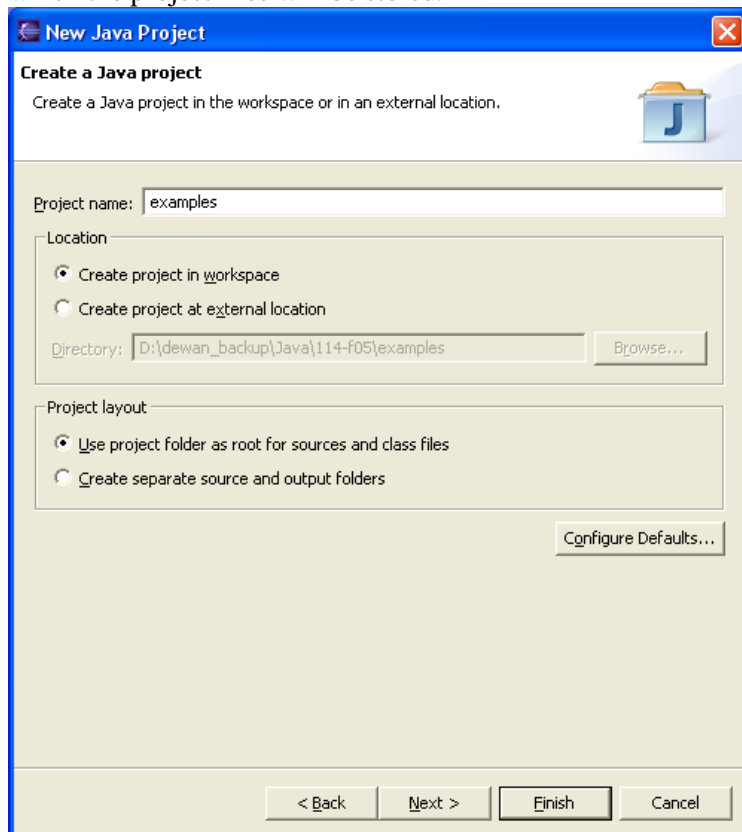
Let execute File→Project to add a new project:



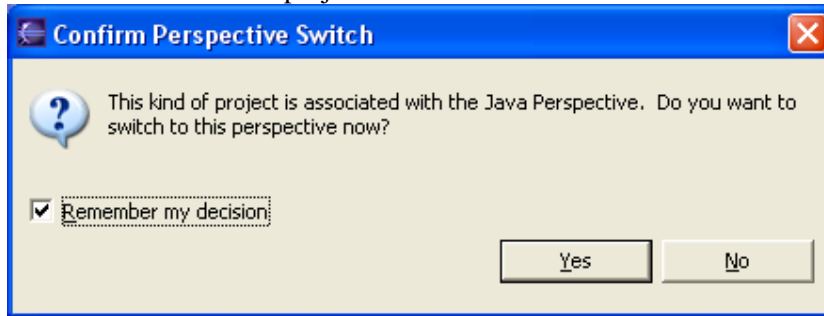
Next we must specify that we want a Java project:



The next step specifies the name of the project, which is also the name of the folder in the workspace in which the project files will be stored.

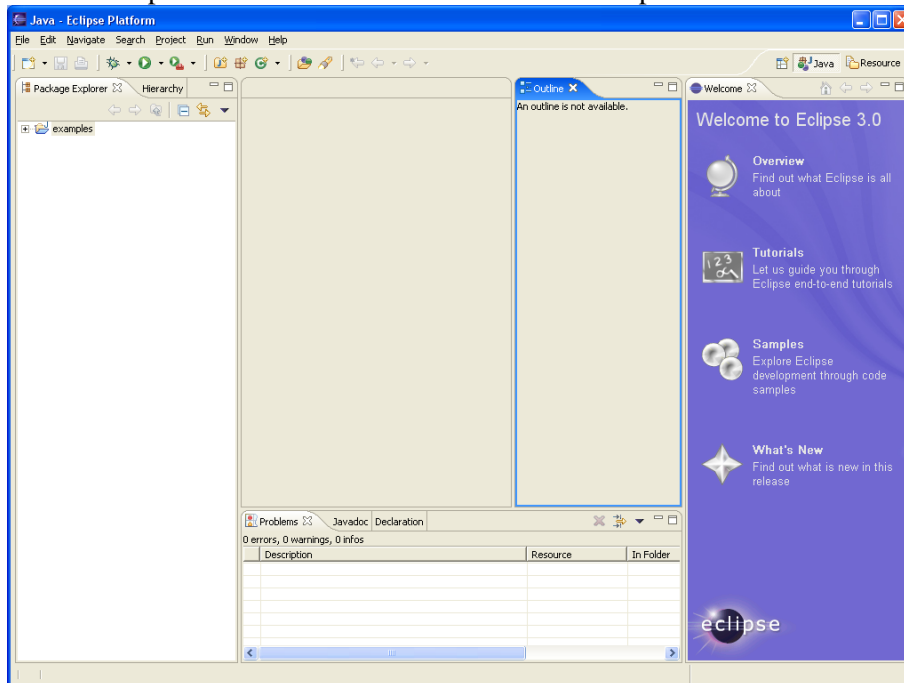


The Finish button takes creates the new project. We then see the following dialogue box, and we say Yes as we have created a Java project and we want to write Java code:

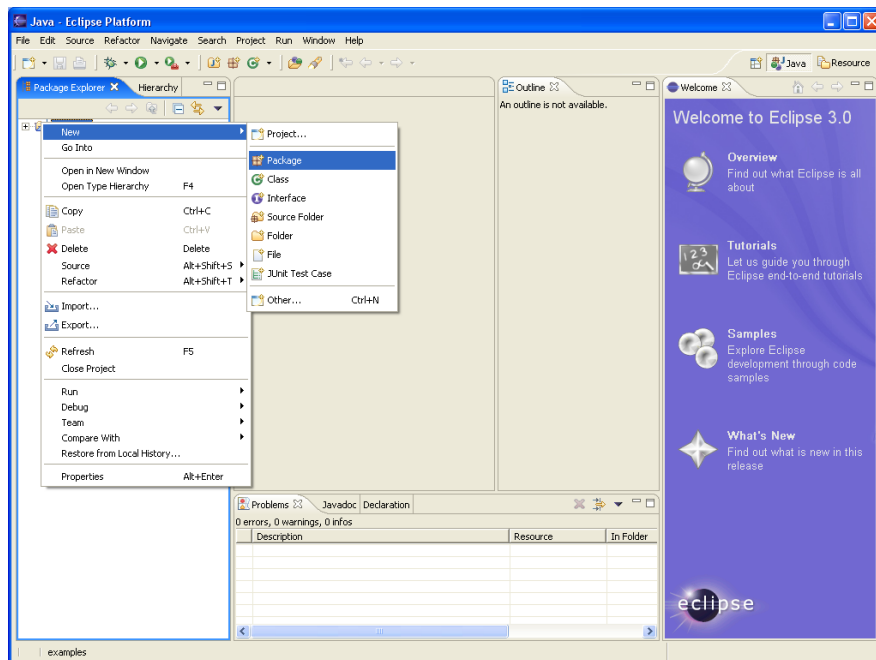


As we will see later, there are kinds of perspectives or views of the programming environment – in particular the Debug perspective. Each perspective is tailored for a particular kind of task. For editing code, the Java perspective is the most ideal.

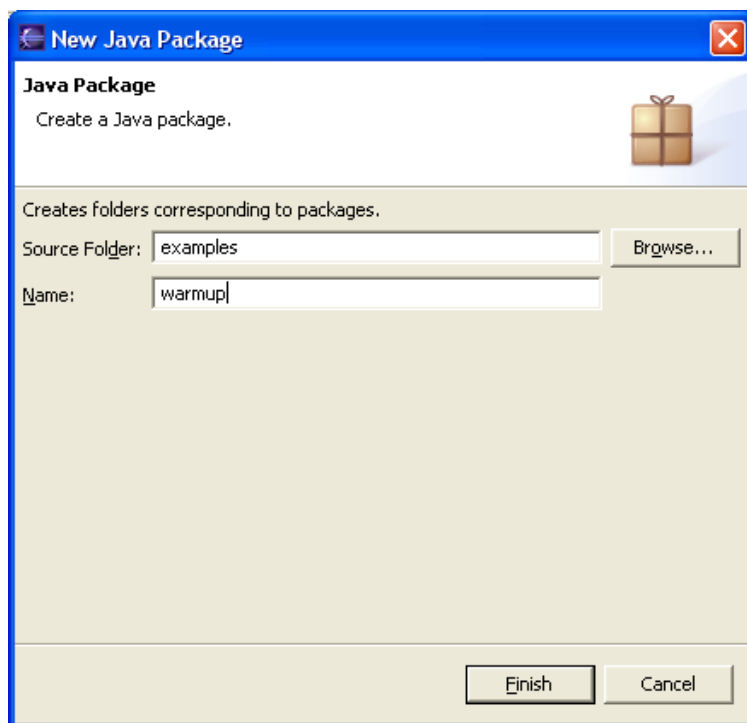
Choosing this perspective takes to the following screen, which essentially tells us an empty Java project called examples has been created in the current workspace.



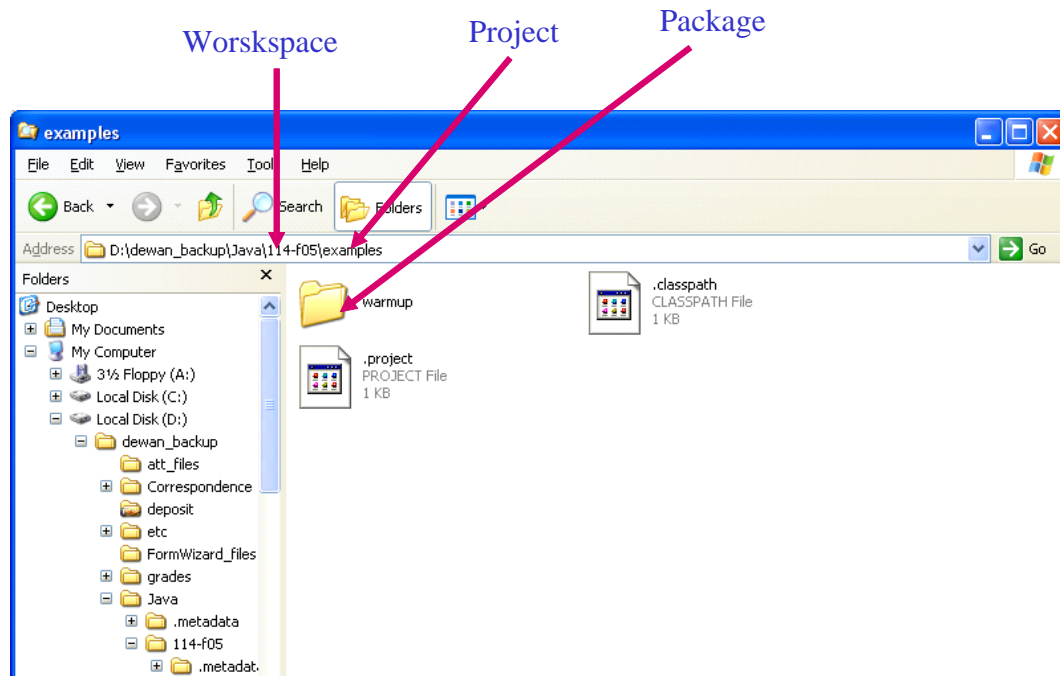
Before we create a new class, we will create the Java package in which it will be put. As this package will be put in the examples project, we will right click on the project and execute the New→Package command



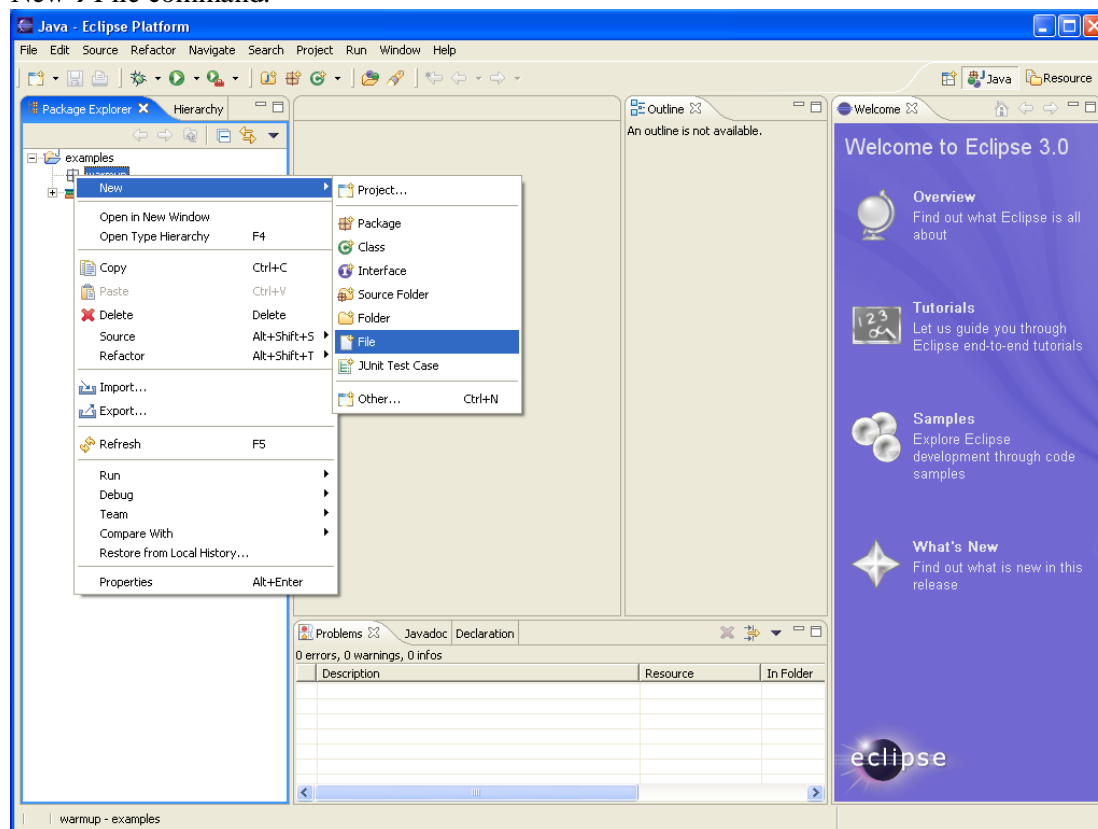
We can now enter the package name:



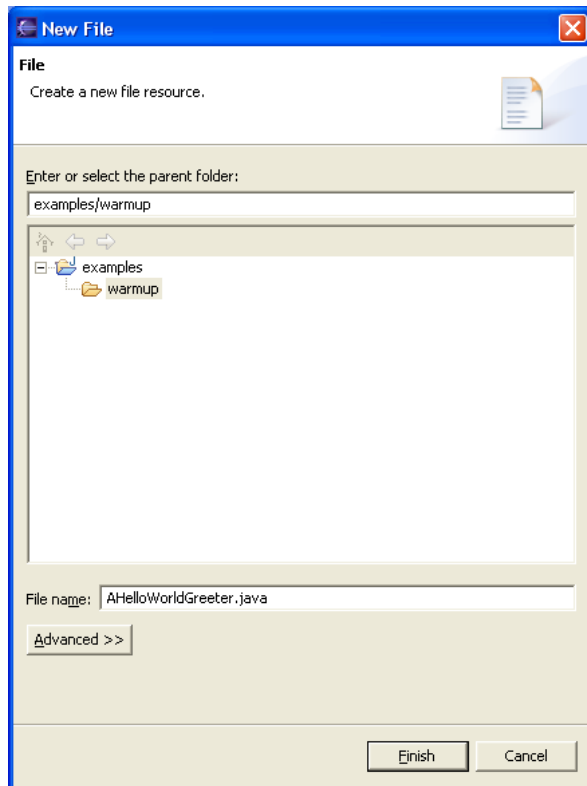
Eclipse creates a new folder for the package, which is nested in the folder it creates for the project, which, recall, is nested in the folder it creates for the workspace, as shown below:



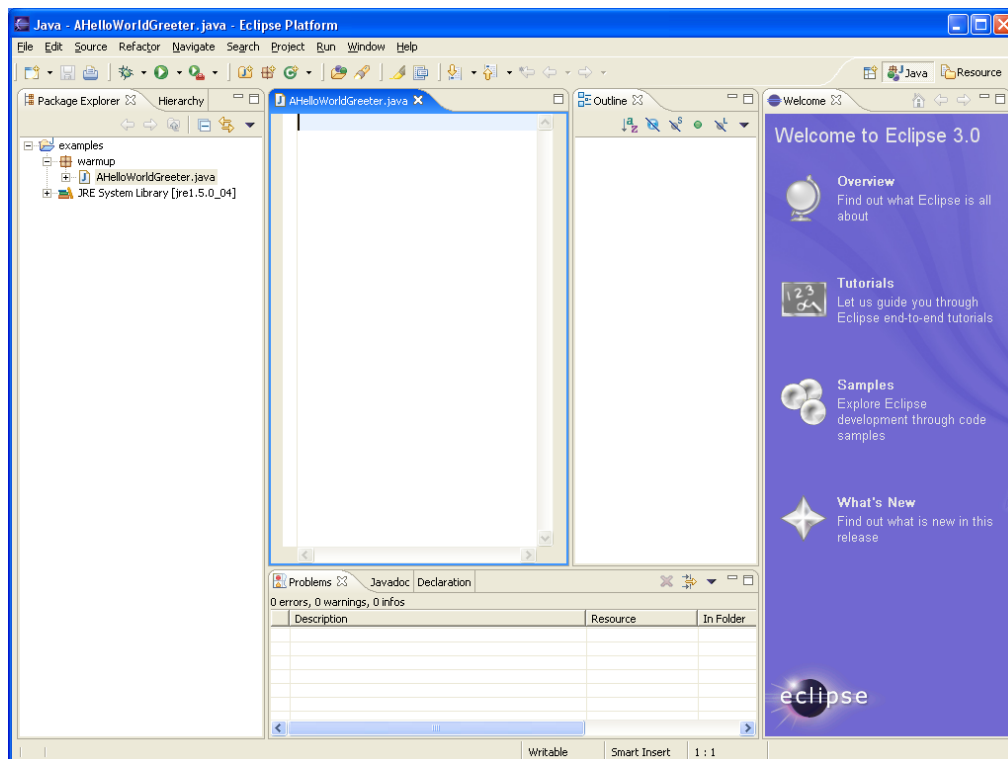
We will now add a new file in the package folder by right clicking on the package and executing the New→File command.



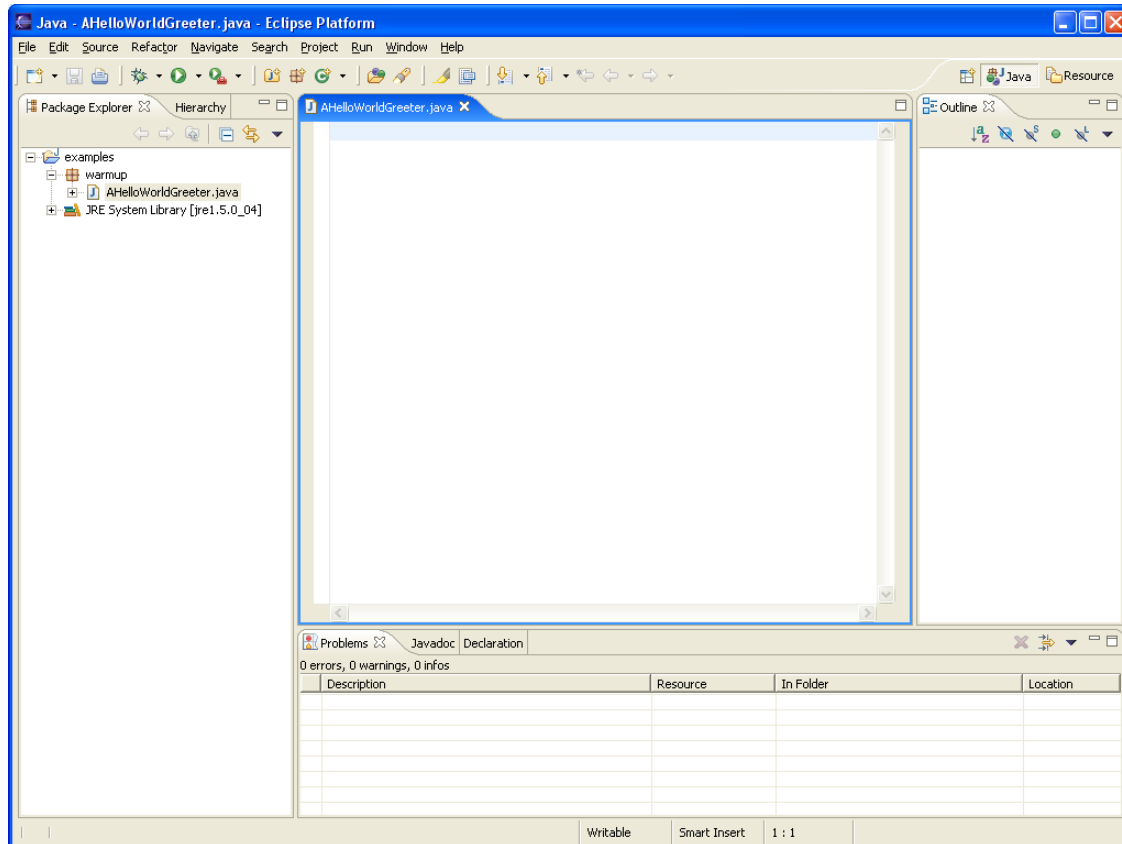
The name of the file must include the suffix: .java to indicate that we will be adding Java source code to the file.



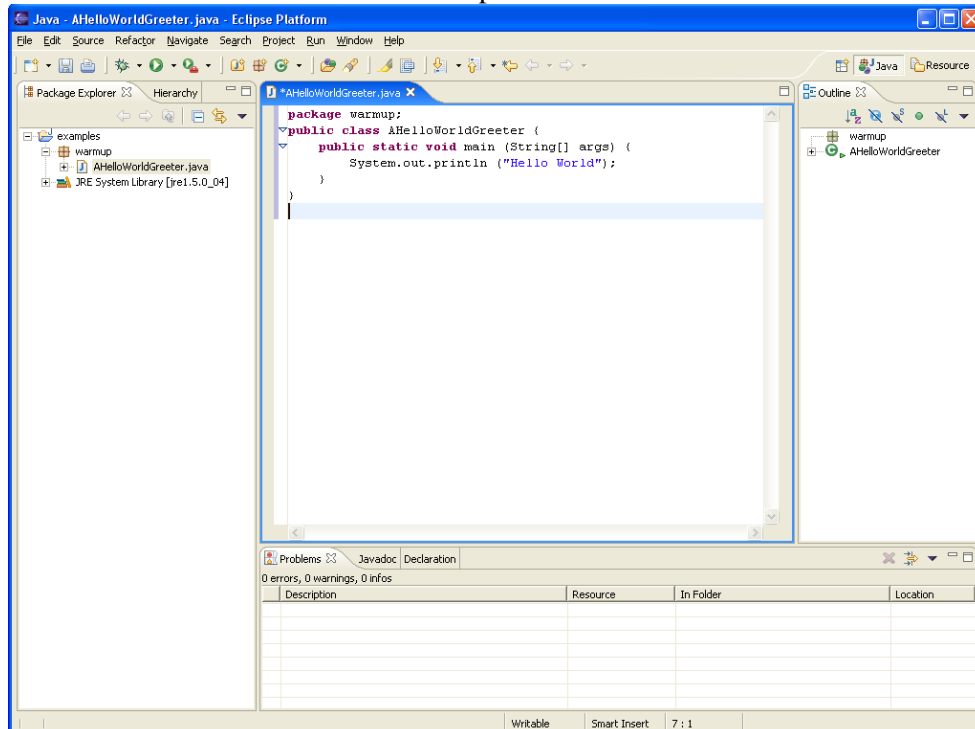
The file name shows up in the explorer view on the left:



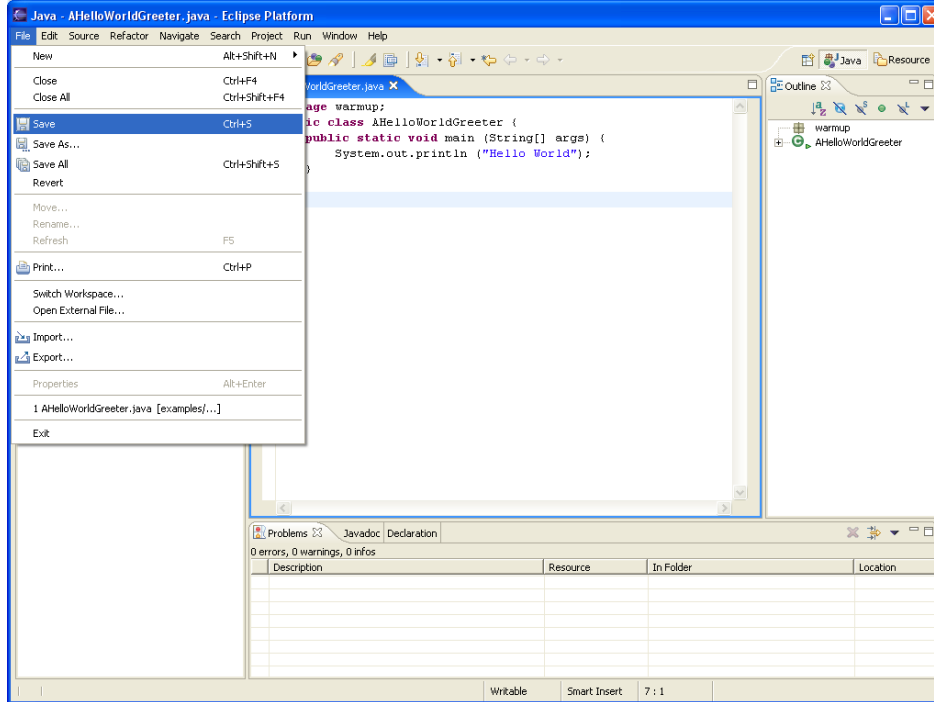
Let us get rid of the welcome pane by clicking on the X decorator:



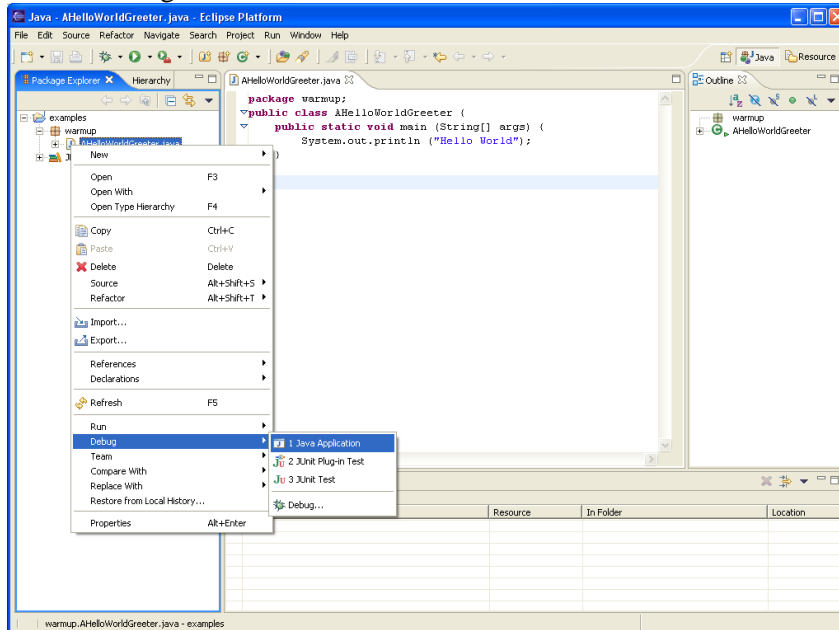
We can now add source code inside the pane created for the new file:



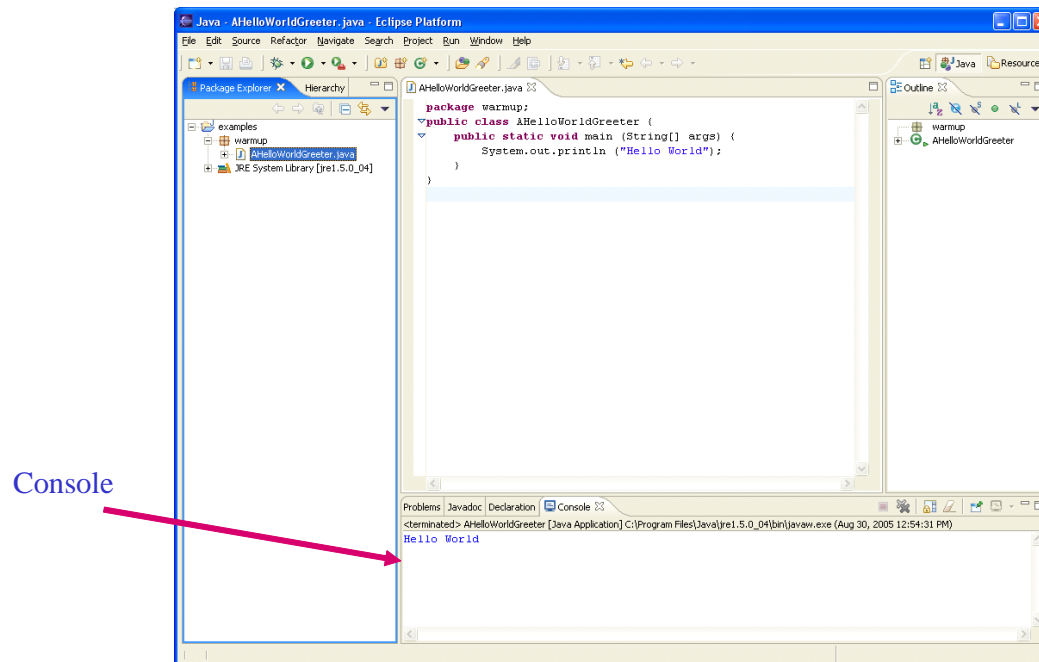
Assuming no errors, we must save the file before running the main method in it:



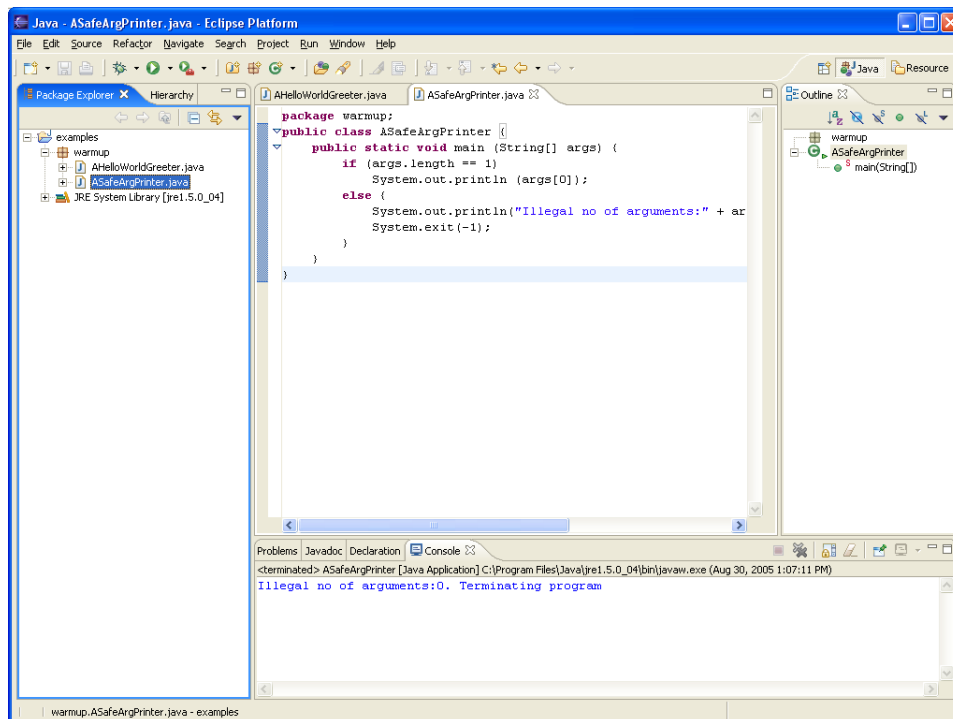
We can only run those classes that have main methods in them. To execute such a class, we must right click on the class name and either execute the Run or Debug command. If we are doubtful if our program actually works, it is better to use the Debug command. The reason is that if the program throws an exception, the debug command will tell us identify the offending line, while the Run command will not. Therefore, we will use the Debug command:



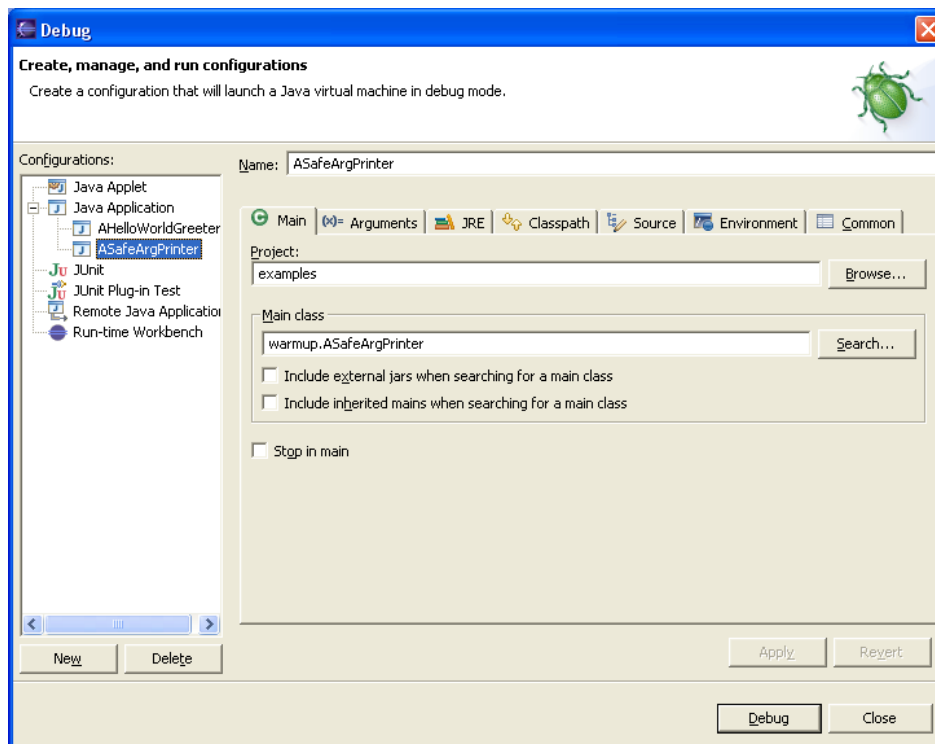
Fortunately, nothing bad happens and we see the correct output in the console window:



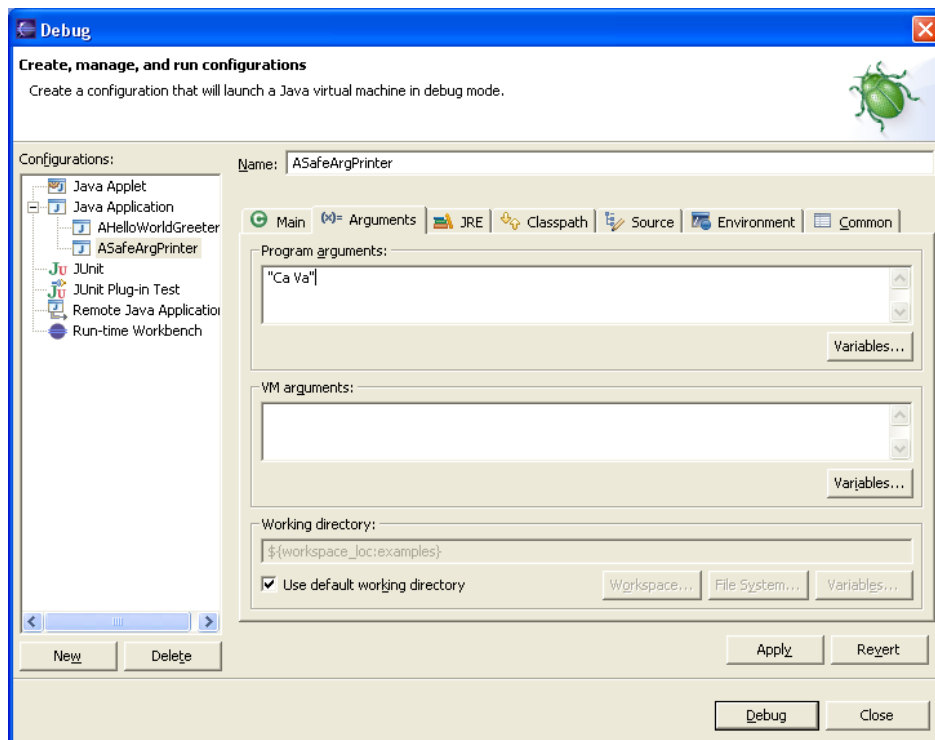
We can go through this process for another program with a main method, ASafeArgPrinter. However, when we run the method, we get an error message from the program:



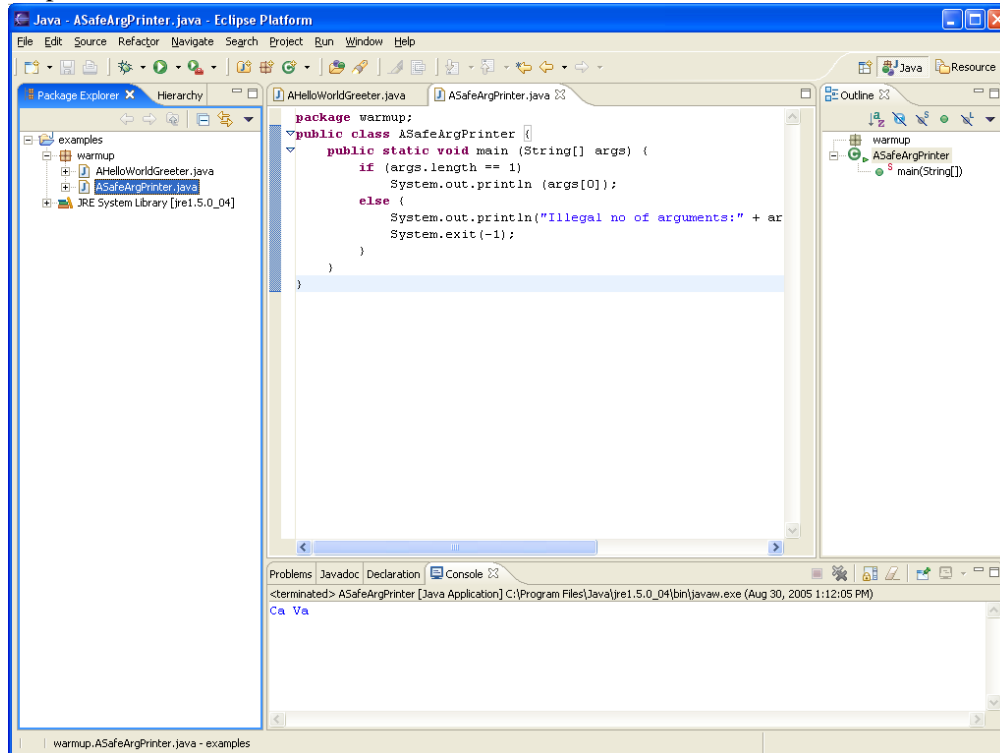
The reason is that we never gave an argument when we ran the program. To do so, we must use the Debug→Debug... command instead of Debug→Java Application:



This command lets us set various aspects of a main class. As we wish to specify its parameters, let us go to the Arguments tab and specify the appropriate Program arguments:



If we now execute Apply, every time we run the main class, these arguments will be used. We can go through this process again to change the arguments. If we now execute Debug, we get the appropriate output:

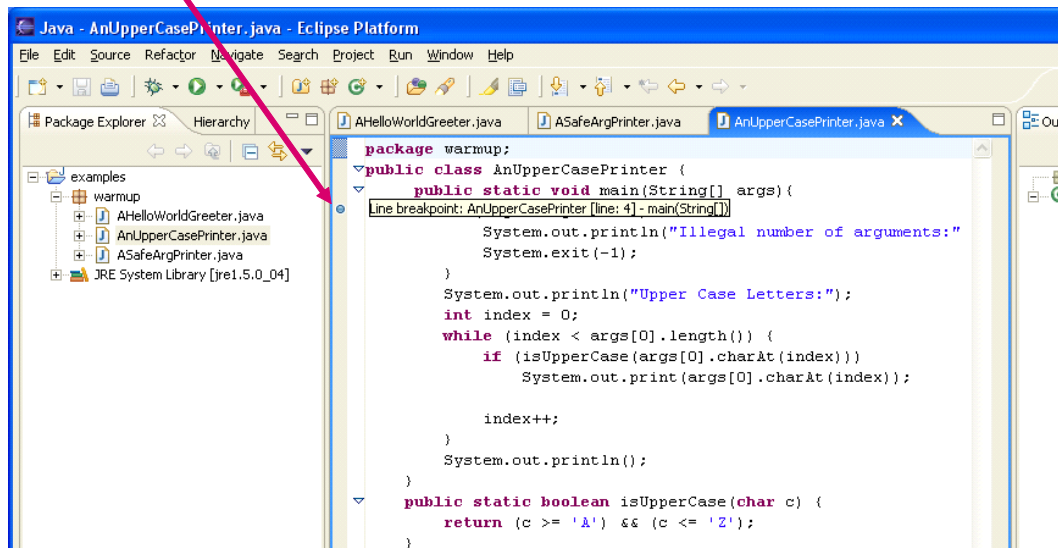


Using the Debugger

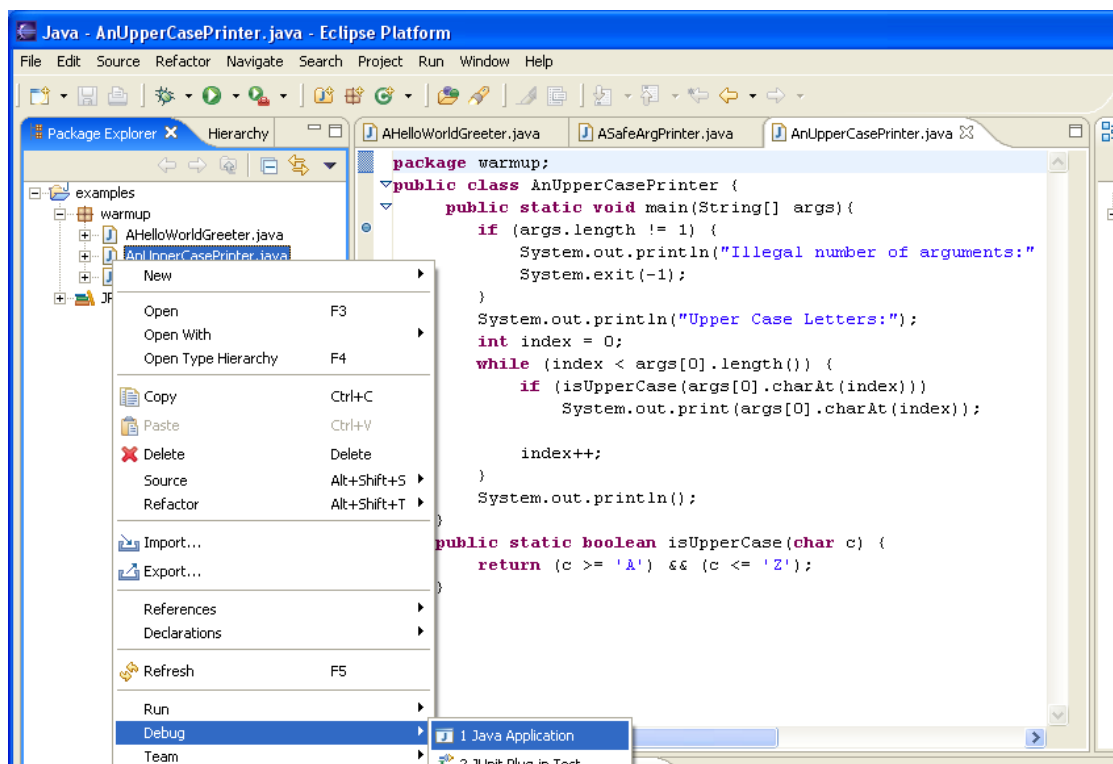
Seldom does one get the right results on the first try, especially for a program of the complexity you will develop in this class. Let us see next, how to use a debugger to run a program.

When we see an error, the first thing we might do is narrow the cause to the start of some statement sequence. We can then set a *breakpoint* at the start of that sequence. We set a breakpoint by simply double clicking next to it in the vertical breakpoint bar. An indication is then given on the bar where we clicked and the statement is highlighted:

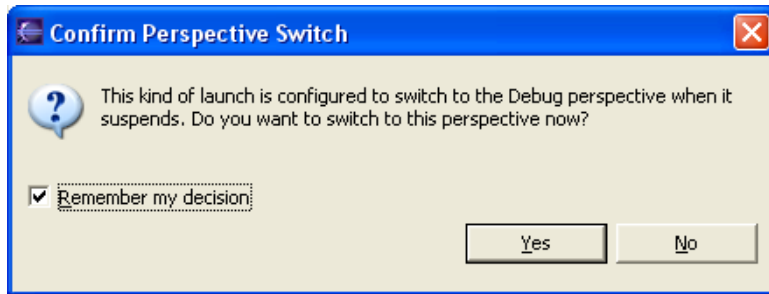
Double click where you want
program to stop



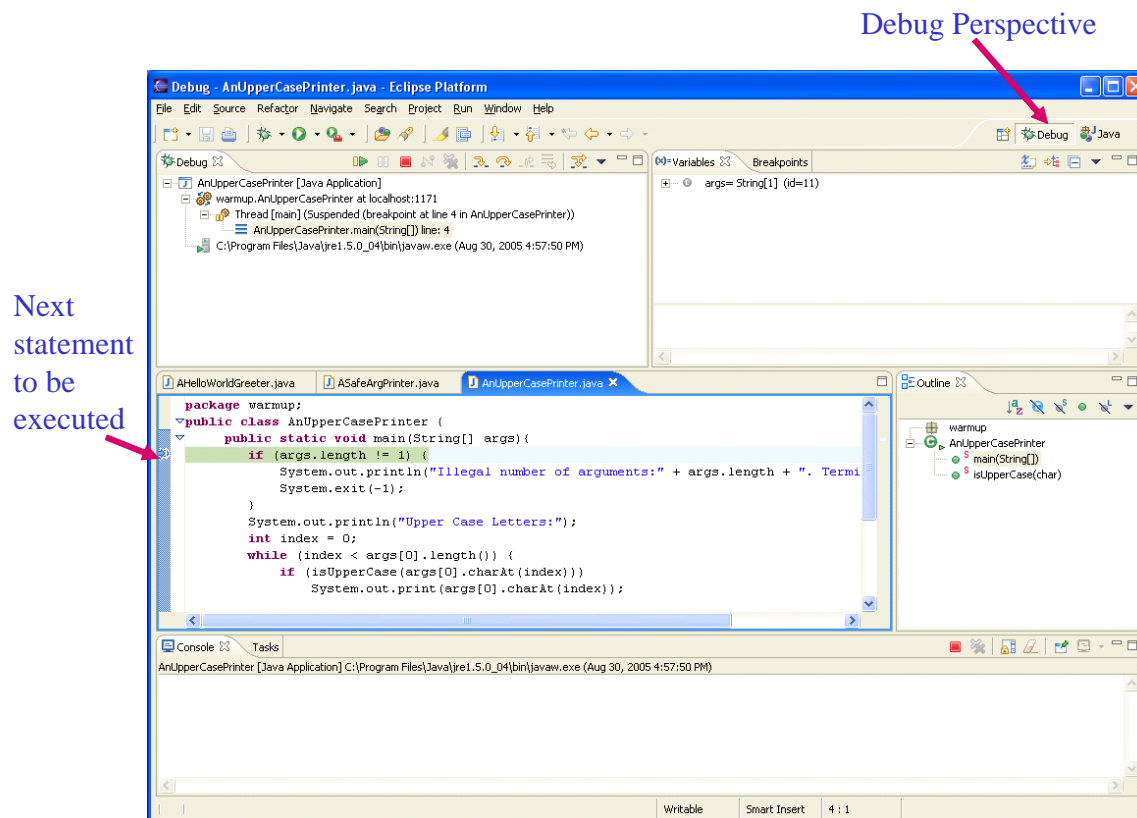
If we now run the program in the debugging mode:



It will ask us if we wish to switch to the Debug Perspective:

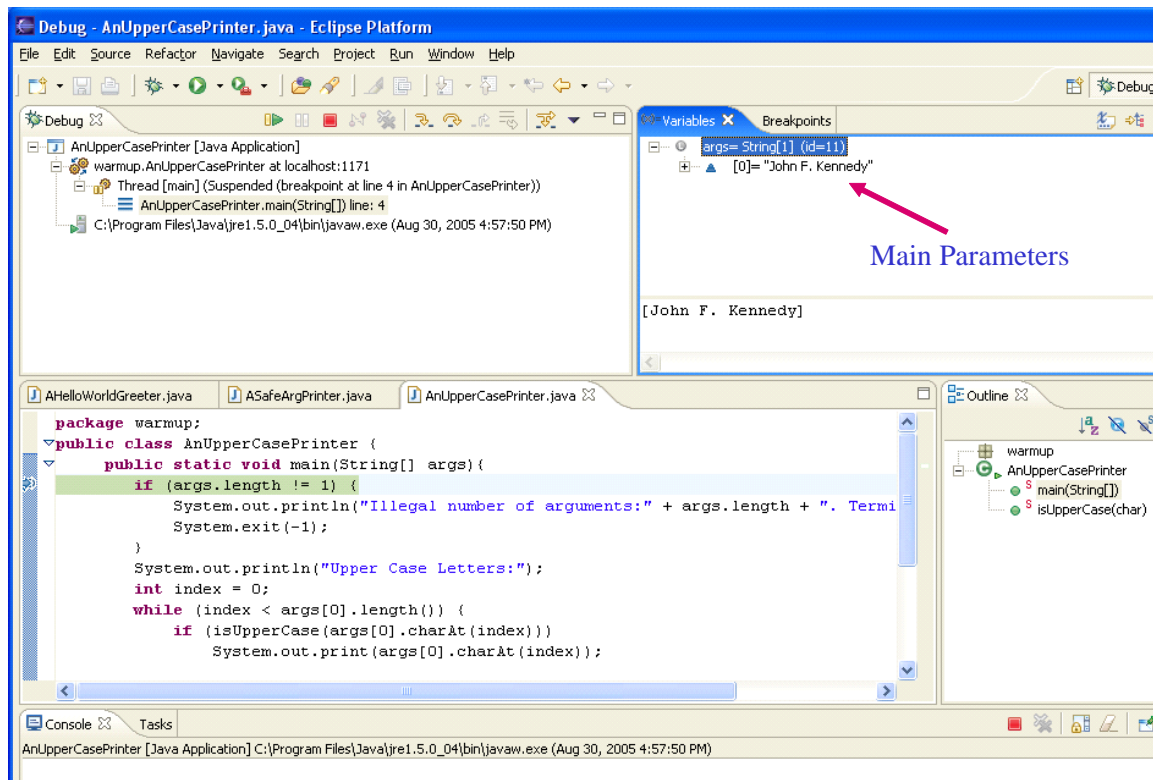


As this perspective gives us the best view for viewing breakpoints, we will say Yes. The program will halt when it reaches the breakpoint:



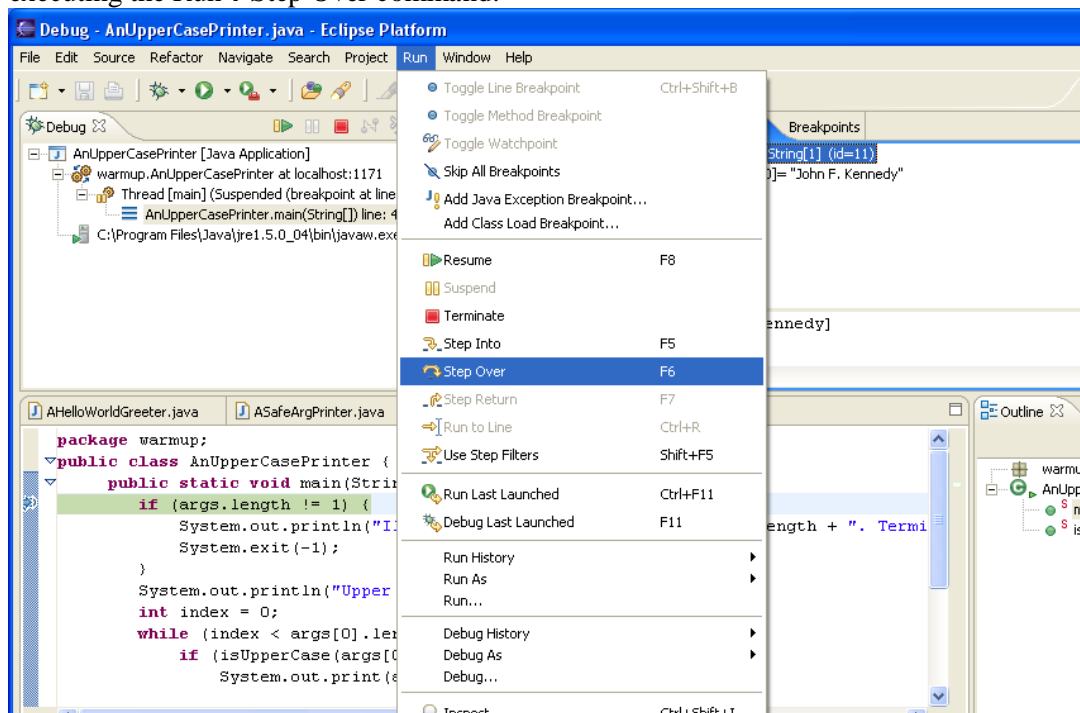
At this point, the next statement to be executed is the one we “broke” into. The arrow next to the statement and its highlight indicate that the Java is about to execute it.

We can examine the values of the variables in the top right window:

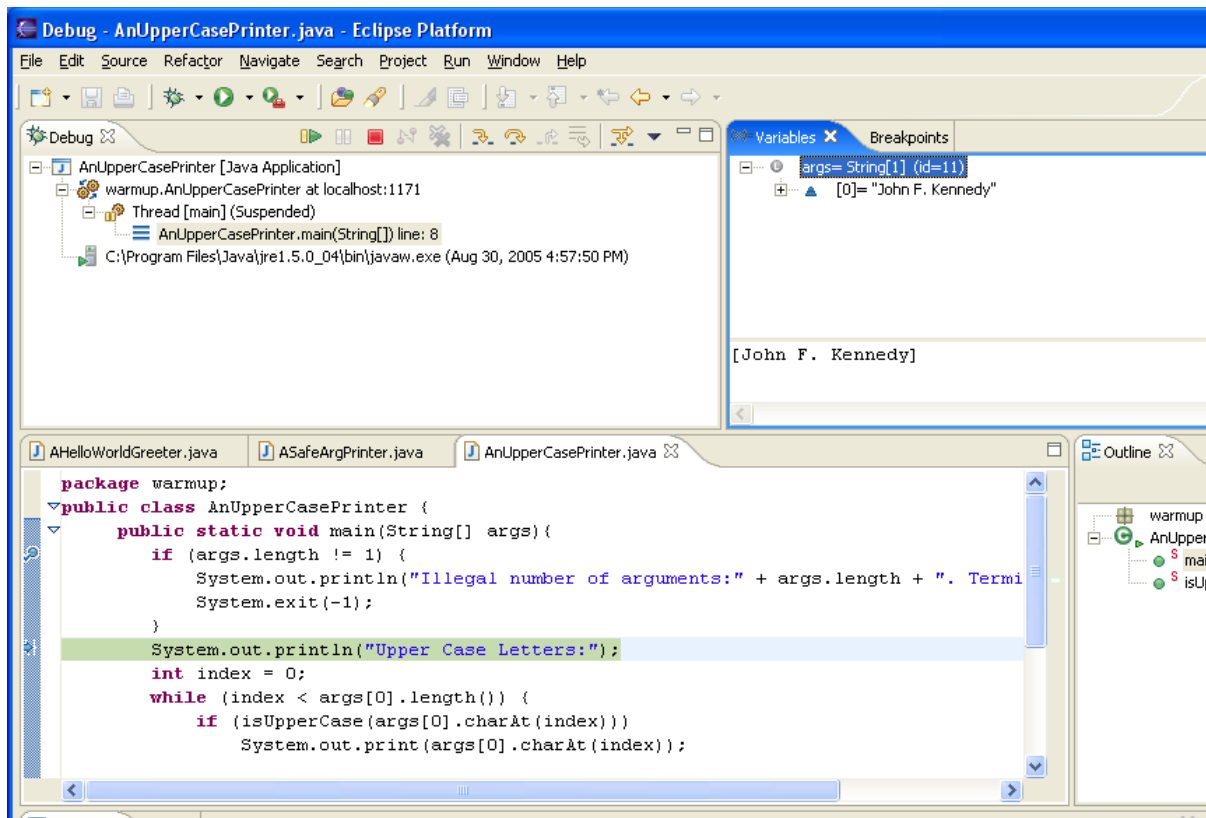


As we see, the string argument we provided has been assigned to the first element of the main parameter, `args`.

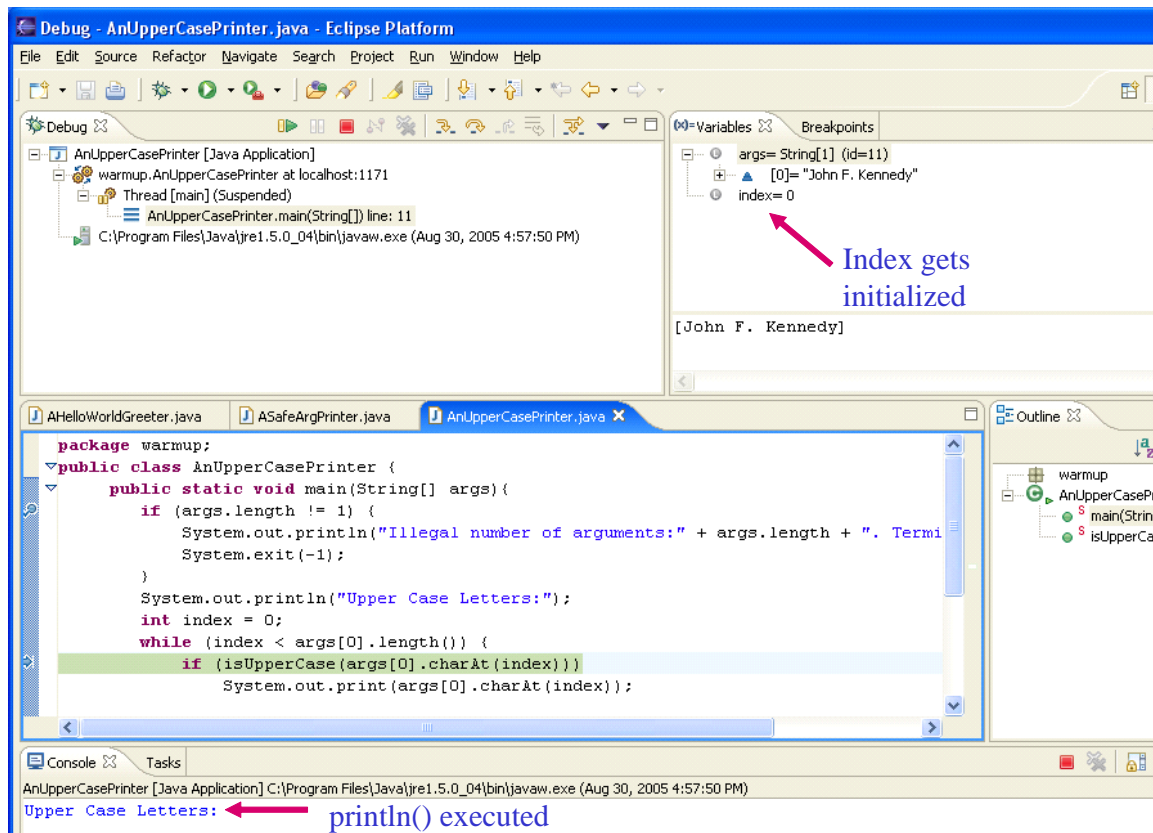
Once we are satisfied that the program state is as we expected, we can break into the next statement by executing the **Run→Step Over** command:



As we see in the figure below, the highlight and the arrow move to the next statement to be executed

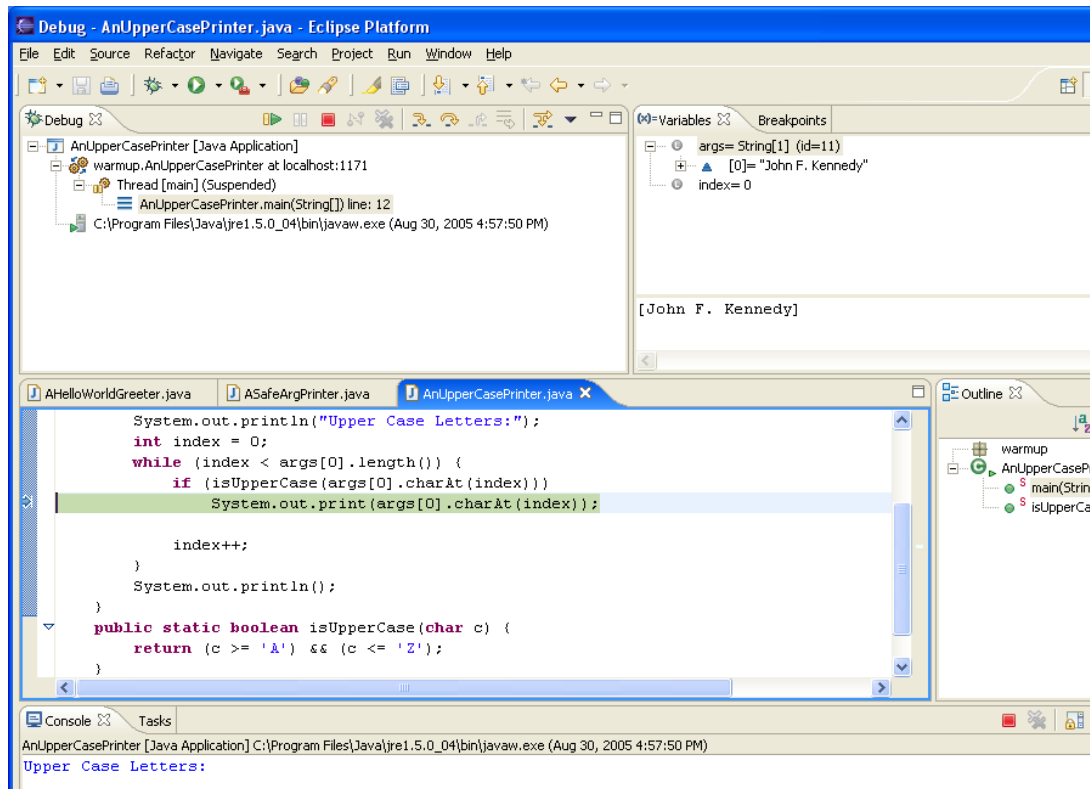


Let us keep executing Run→Step Over (the F6 key is an accelerator for it) until we reach the following statement:

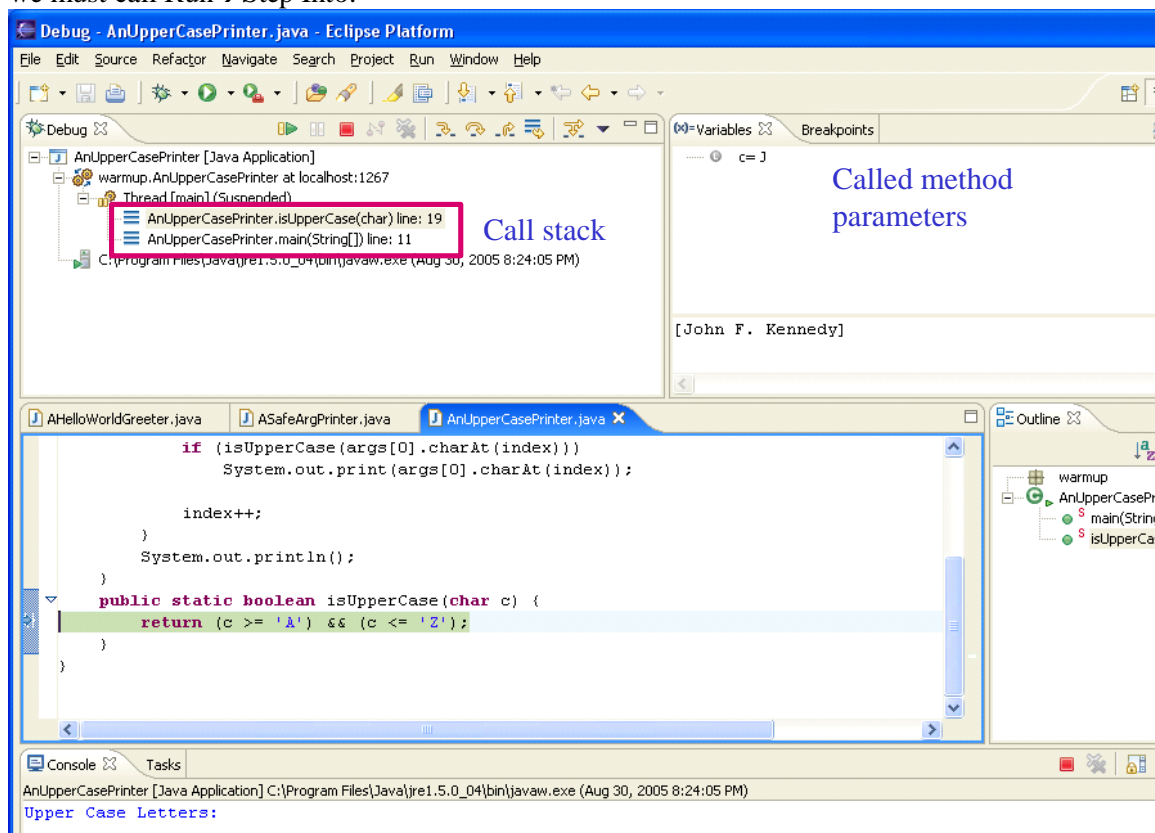


At this point, the console contains the output produced by the `println()` call we have stepped over and thus finished executing. We also see in the Variables window an entry for `index`. The value of the variable is `0` at this point, as this is the first time we are going through the loop. If we keep single stepping, we can see how many times the loop is executed.

What should happen if we execute `Run` → `Step Over` at this point to go to the next statement? Should it go to the next statement in the `main` method or first statement in the `isUpperCase()` method? As shown below, the answer is the former:

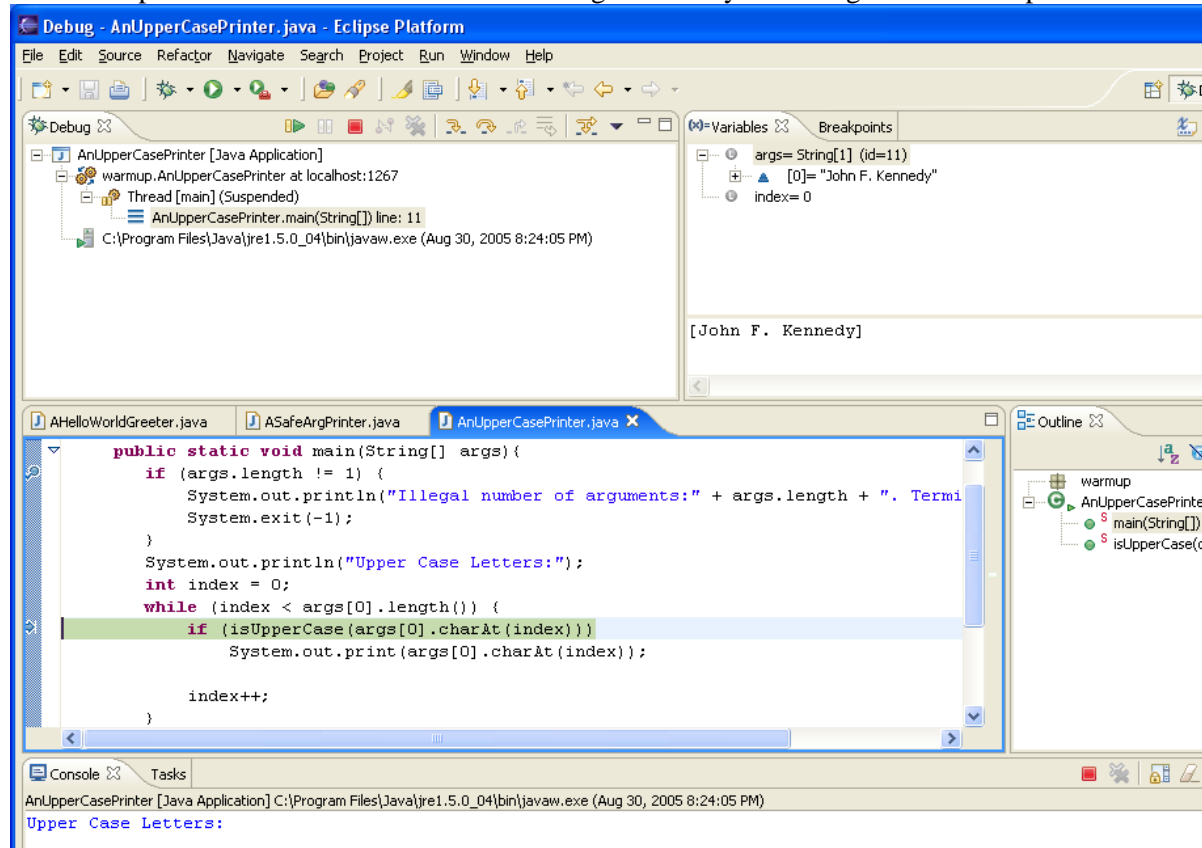


The reason is that Run→Step Over steps over any called method. If we want to step into the called method, we must call Run→Step Into:

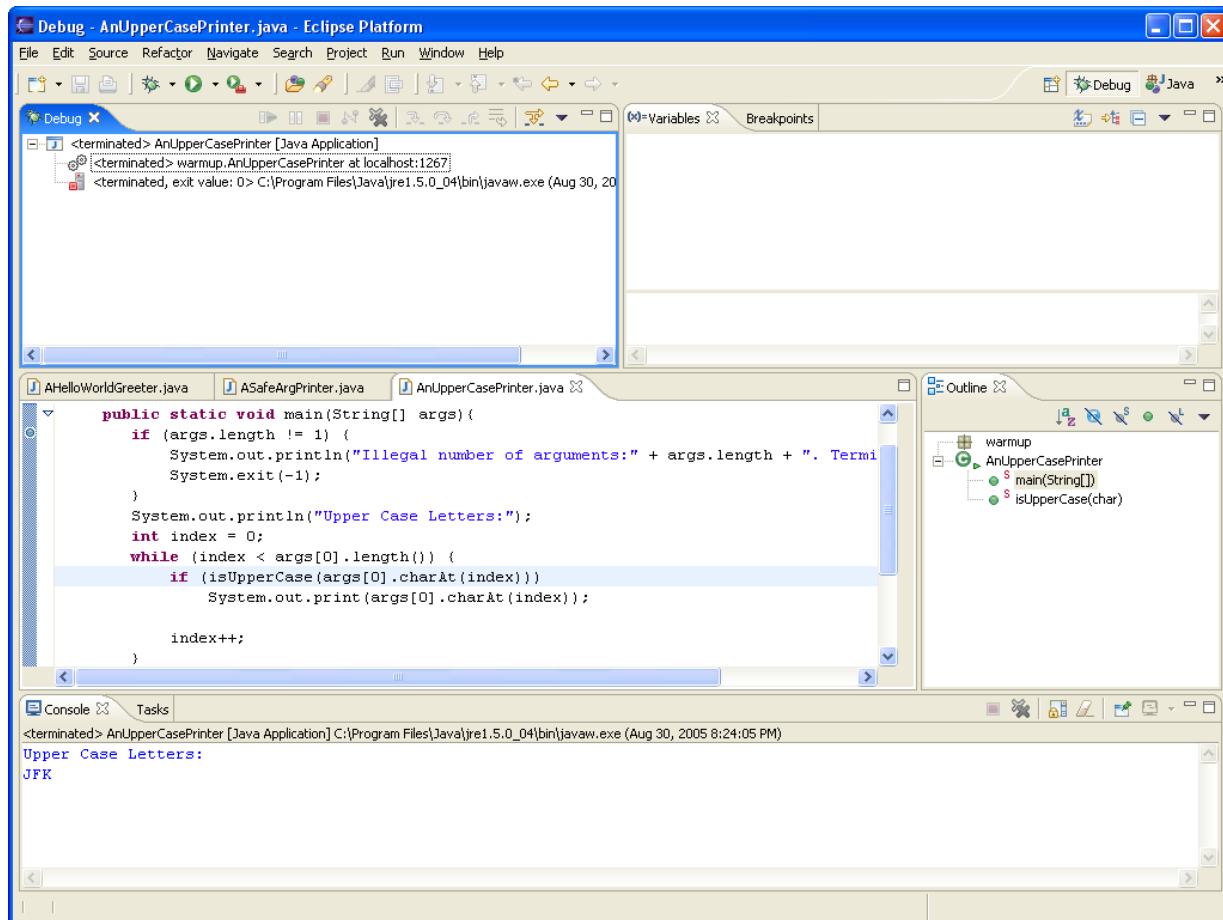


As the top left Stack window shows, we have two pending or stacked calls, the call to `main()` made by the interpreter, and the call to `isUpperCase()` made by `main()`. The Variables window now shows the variables of the called method, telling us that the `C` char parameter is initialized to the letter 'J'. If we wish to see the variables of the main method, we can select it in the call stack, which will restore the contents of the Variables window shown earlier.

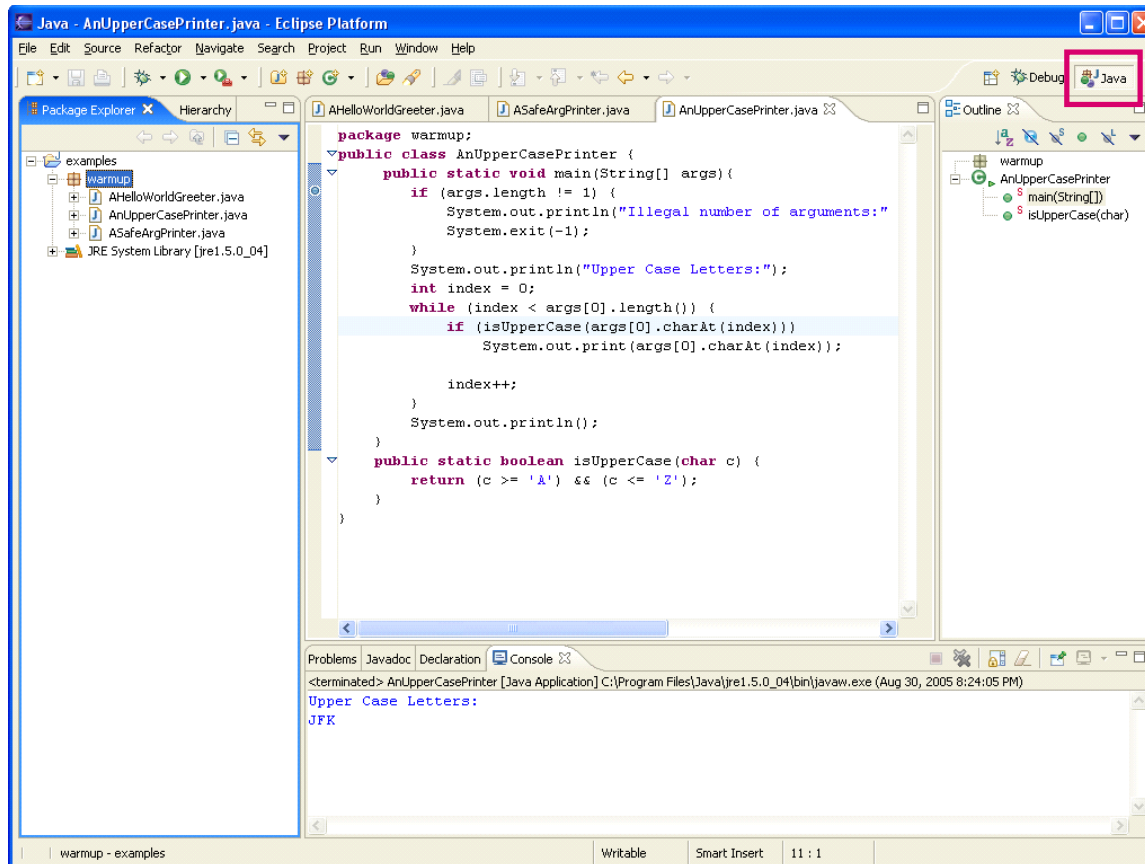
We can step out of the called method to the calling method by executing the **Run→Step Return** command:



To resume normal execution where we don't step through the program statements, we execute the **Run→Step Resume** command. The program quickly completes and shows us the output:

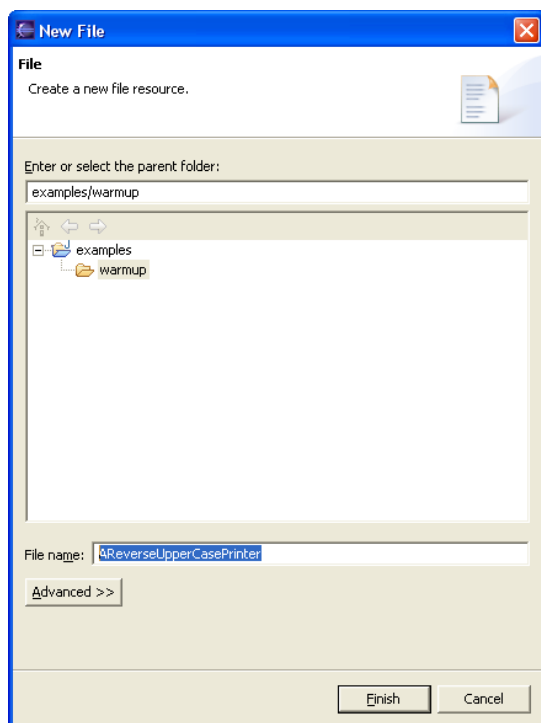


Recall that we are currently in the Debug Perspective. If we wish to start editing code again, we can switch to the Java Perspective by pressing on the Icon for it in the upper right corner of the window:

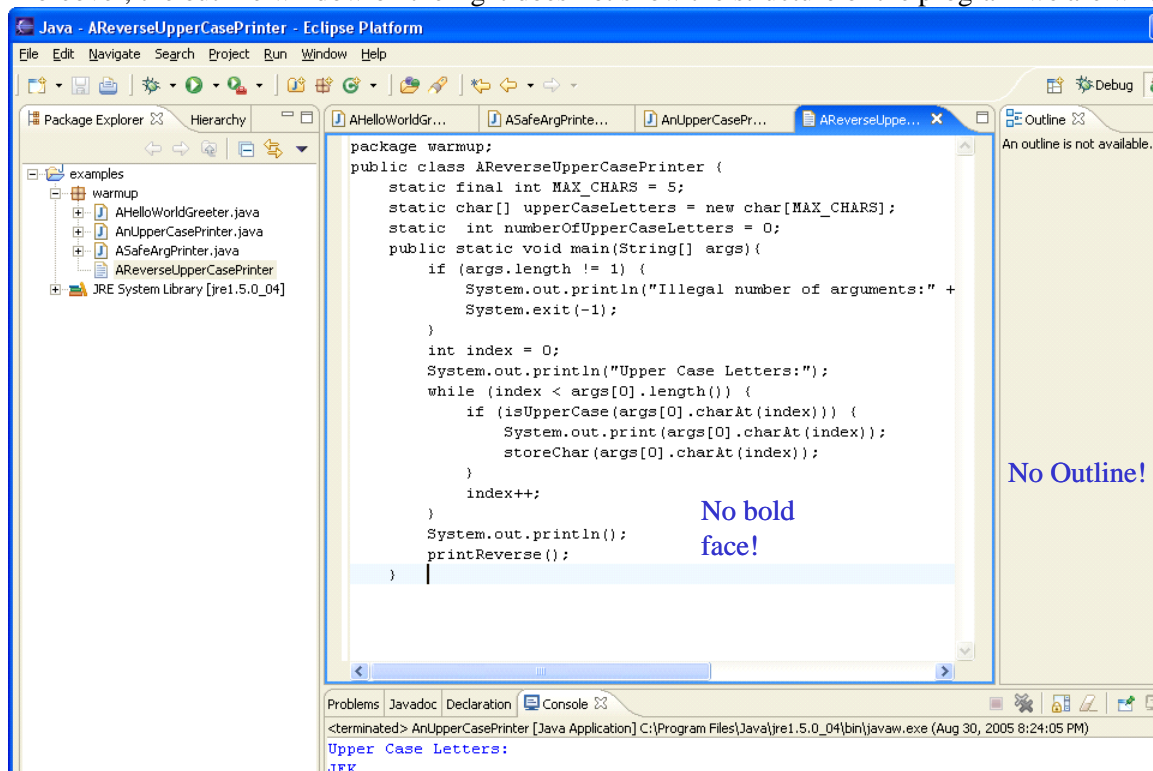


Refactoring

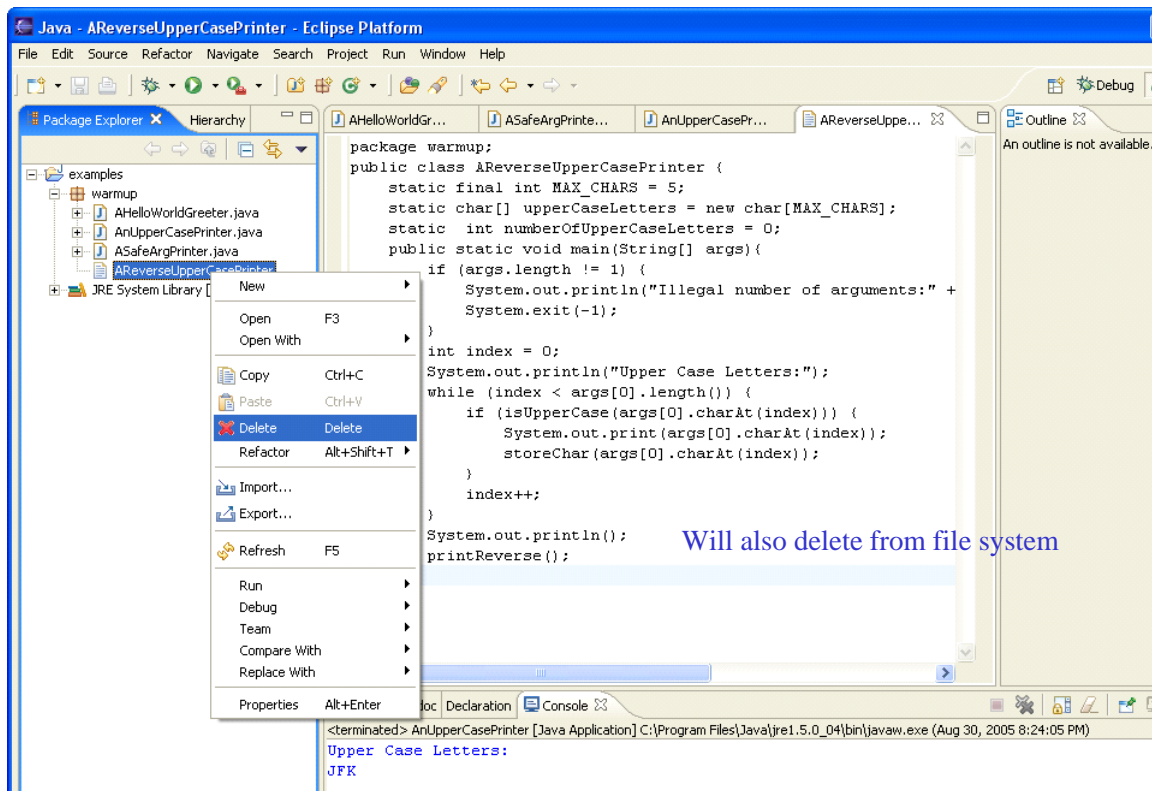
Let us say we wish to add another class to the package. As before we can right click on the package name, execute the New→File command, and fill in the name of the class:



However, things are different this time. The keywords code we add to the file do not, however, get bold. Moreover, the outline window on the right does not show the structure of the program we are writing:

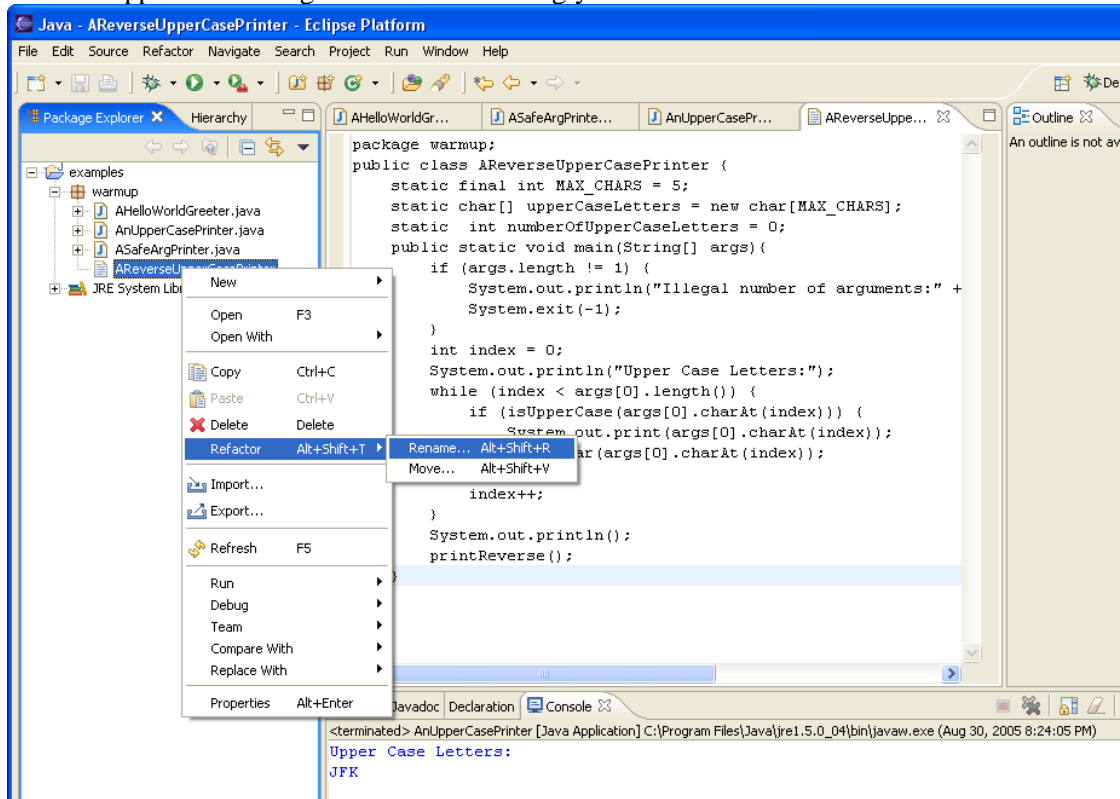


The reason is that we forgot to add the .java suffix to the file name. Therefore, Eclipse does not realize that we are writing Java code. At this point, we could delete the file:

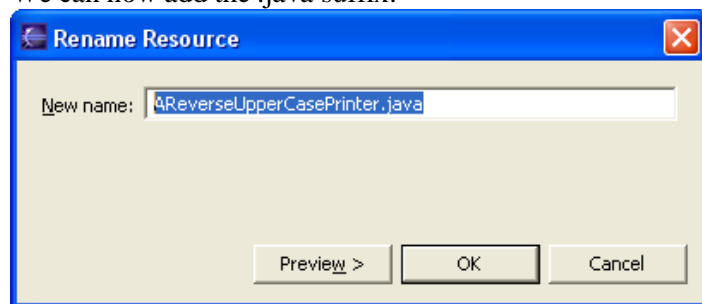


But this would mean we have to redo our work. (In some other environments, deleting a file from a project does not remove it from the file system. In Eclipse, it is also removed from the file system.)

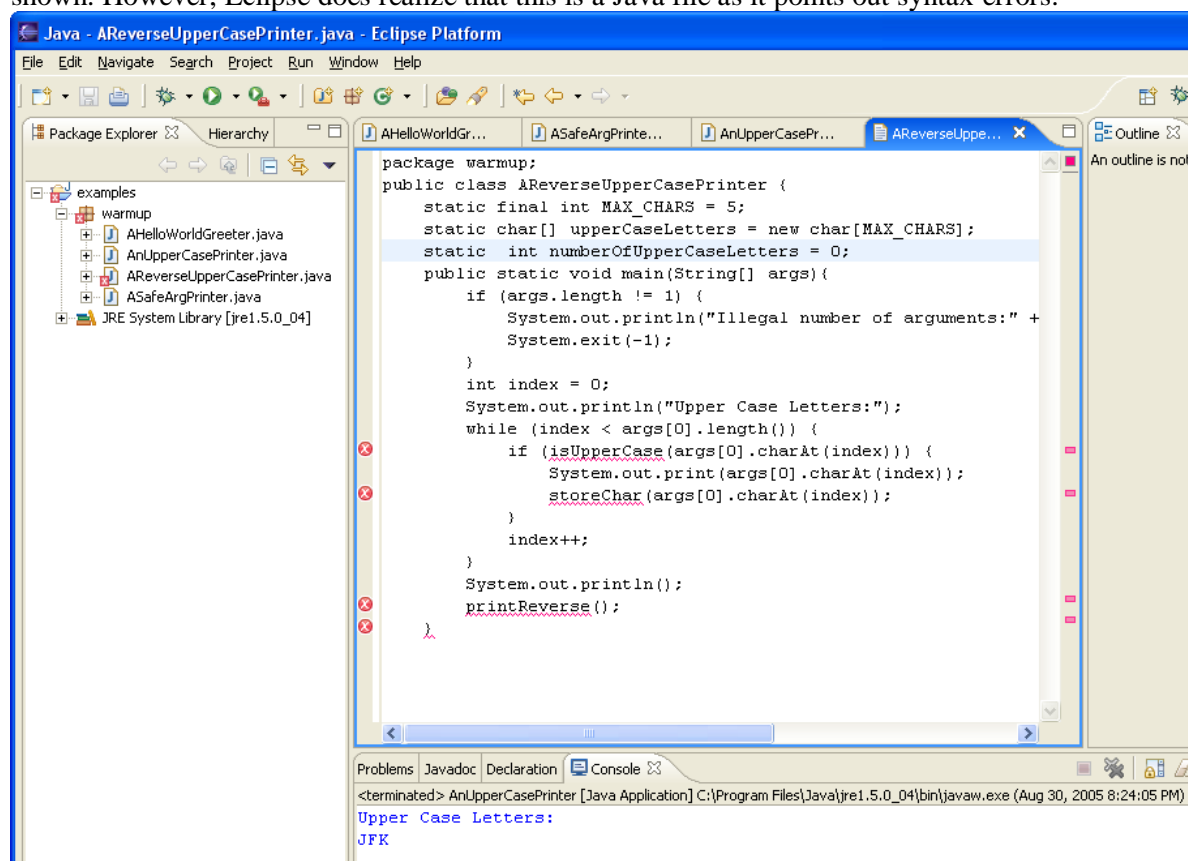
A better approach is to right-click on the wrongly named file and execute the Refactor→Rename command:



We can now add the .java suffix:



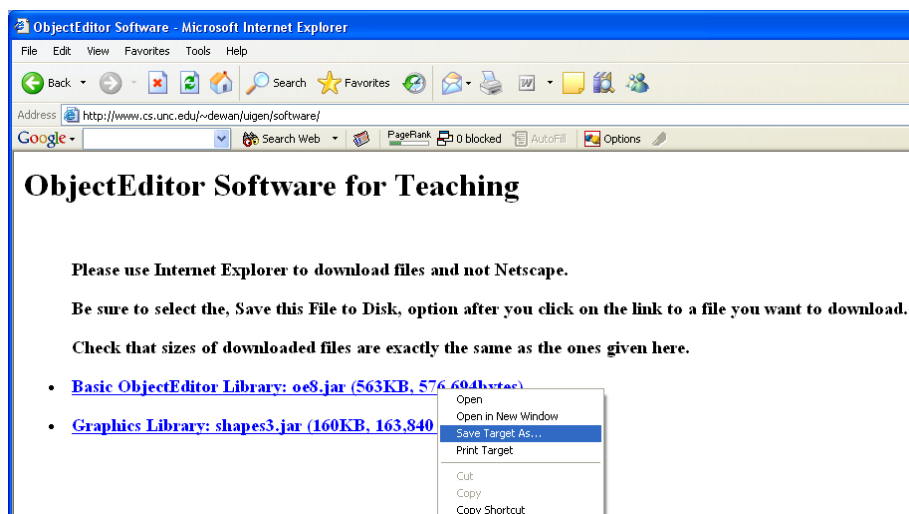
In the version of Eclipse I used, the keywords still remain unboldened. Moreover, the outline view is still not shown. However, Eclipse does realize that this is a Java file as it points out syntax errors:



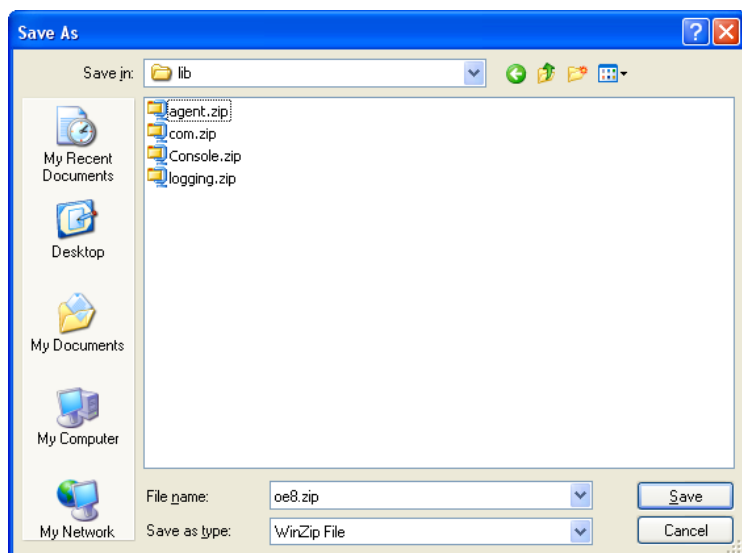
Installing a Library

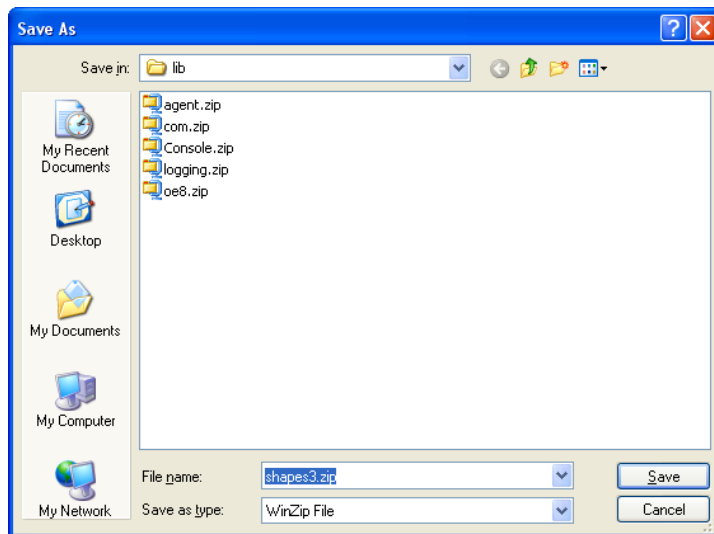
In this course, we will be using a system called ObjectEditor, which was developed by my research group. To use it, we must tell our programming environment to make it part of the library used in our project. The following screenshots illustrate this process for ObjectEditor.

Before we create a library, we must load the code for it on our computers. In case of ObjectEditor, the code is stored in two files: oe.jar and shapes.jar, which can be obtained from:
<http://www.cs.unc.edu/~dewan/uigen/software:>



As we see above, there are two libraries to be downloaded: oe8.jar and shapes3.jar. Let us load them in a special directory that keeps our Java libraries:

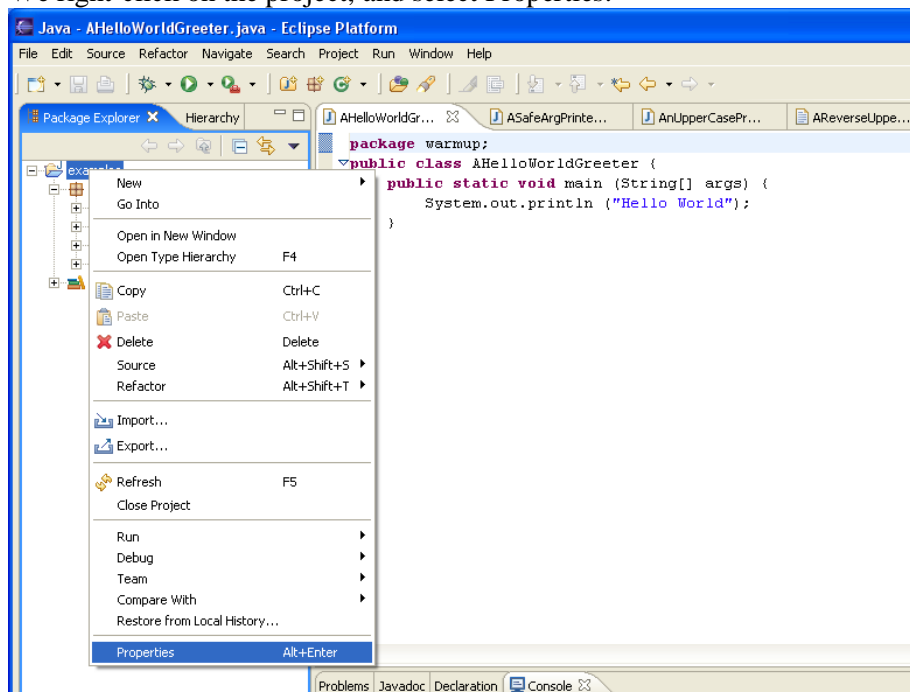




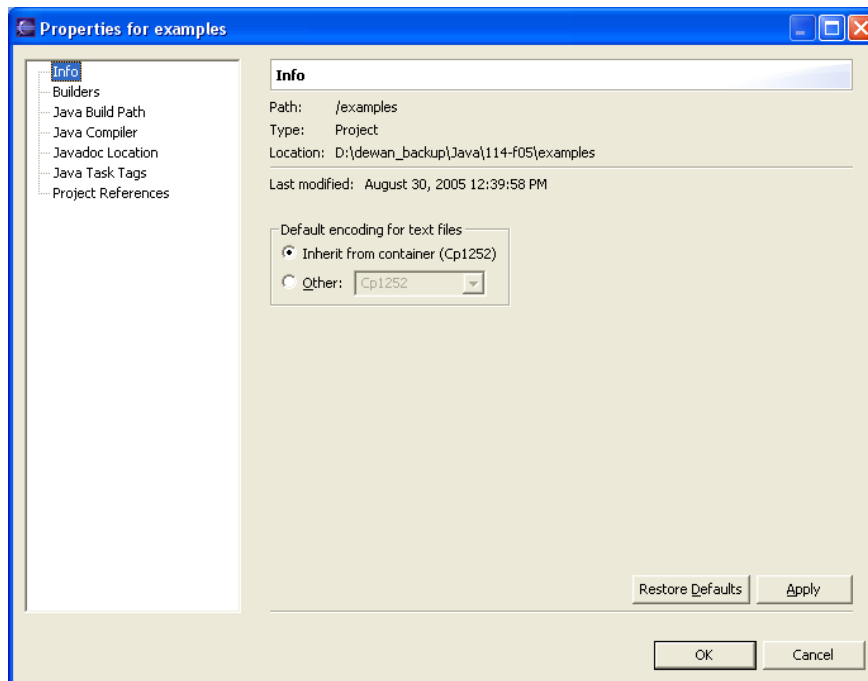
Notice that the browser prompts us to store them as zip files rather than jars. This is because the two are equivalent. On a Windows platform it is better to use the suffix .zip for these files because WinZip can then recognize them.

Now we must tell Eclipse about them. Each project can refer to a different set of libraries, so we must link these two libraries to each project that uses ObjectEditor.

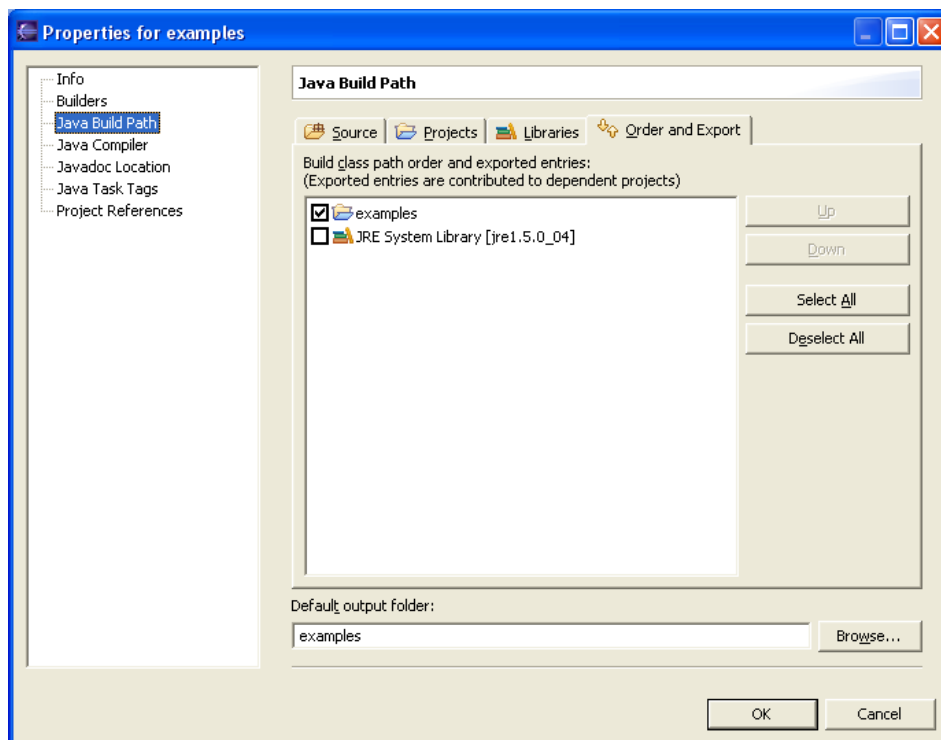
We right-click on the project, and select Properties:



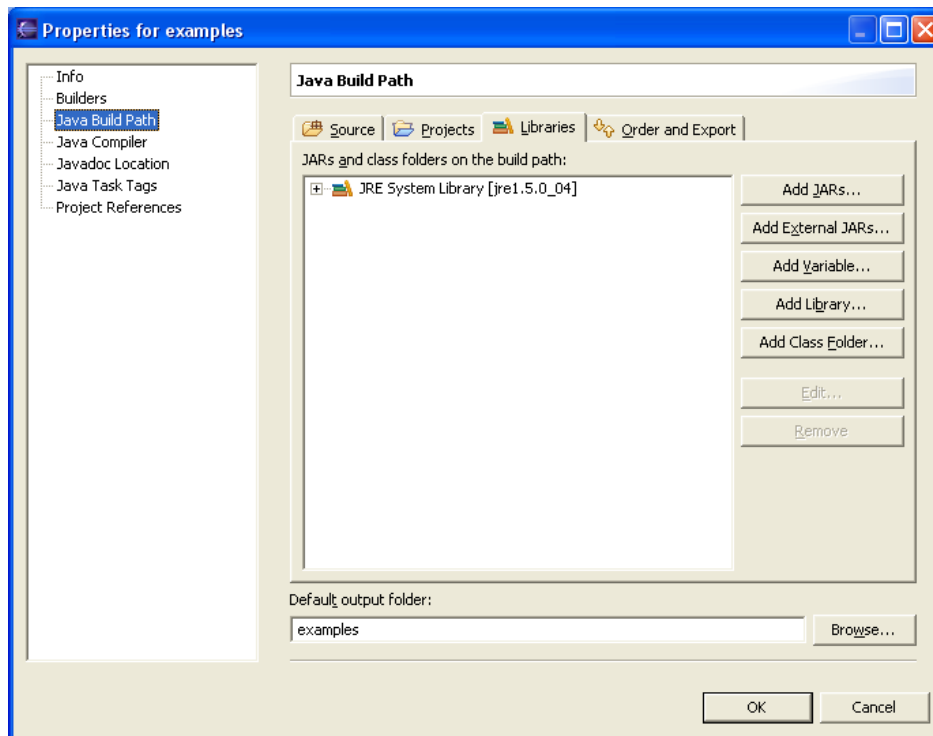
This results in the following screen:



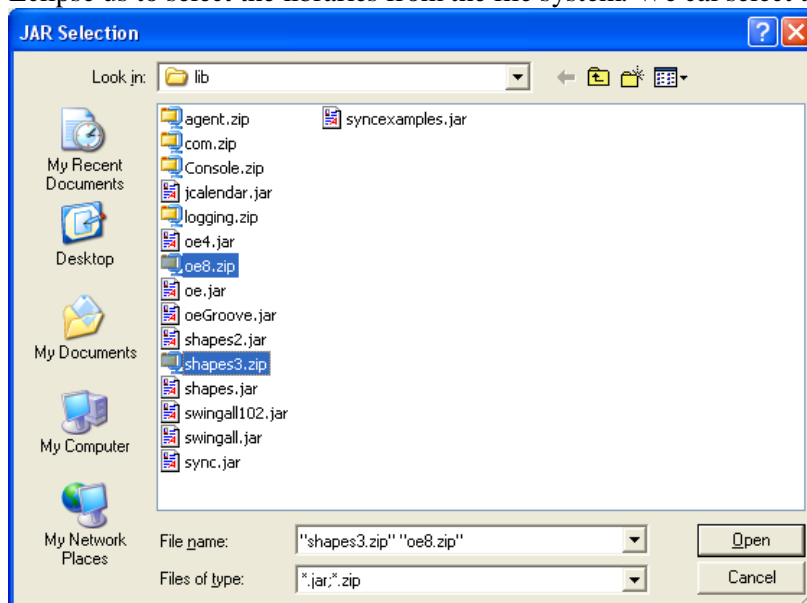
We select Java Build Path:



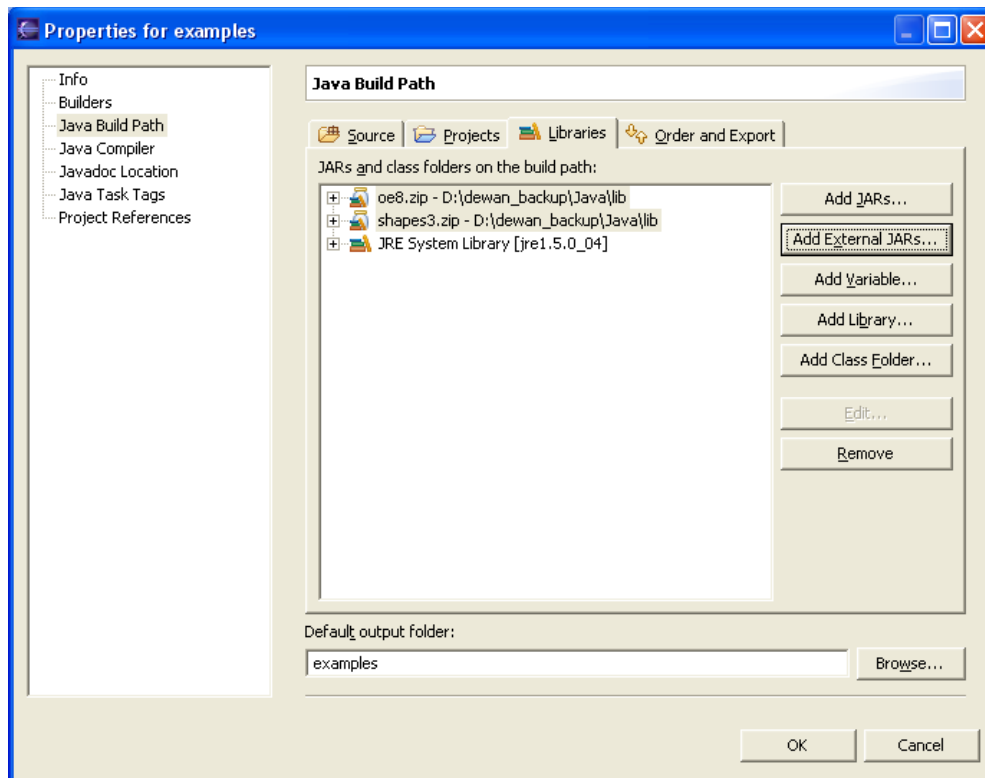
The next step is to select the library tab.



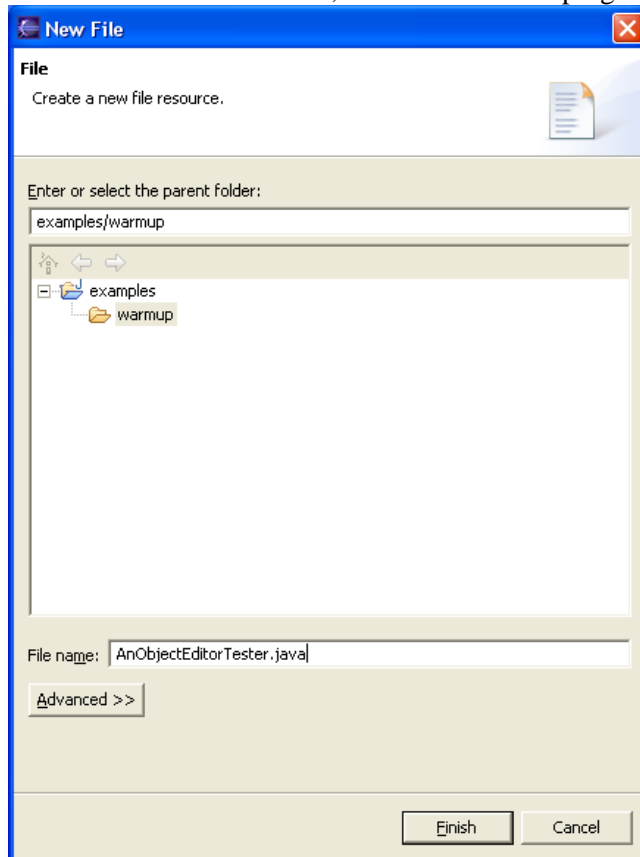
As the jar files we are about to add are in an external directory, we press the Add External JARs button. Eclipse us to select the libraries from the file system. We cal select the two jar/zip files.



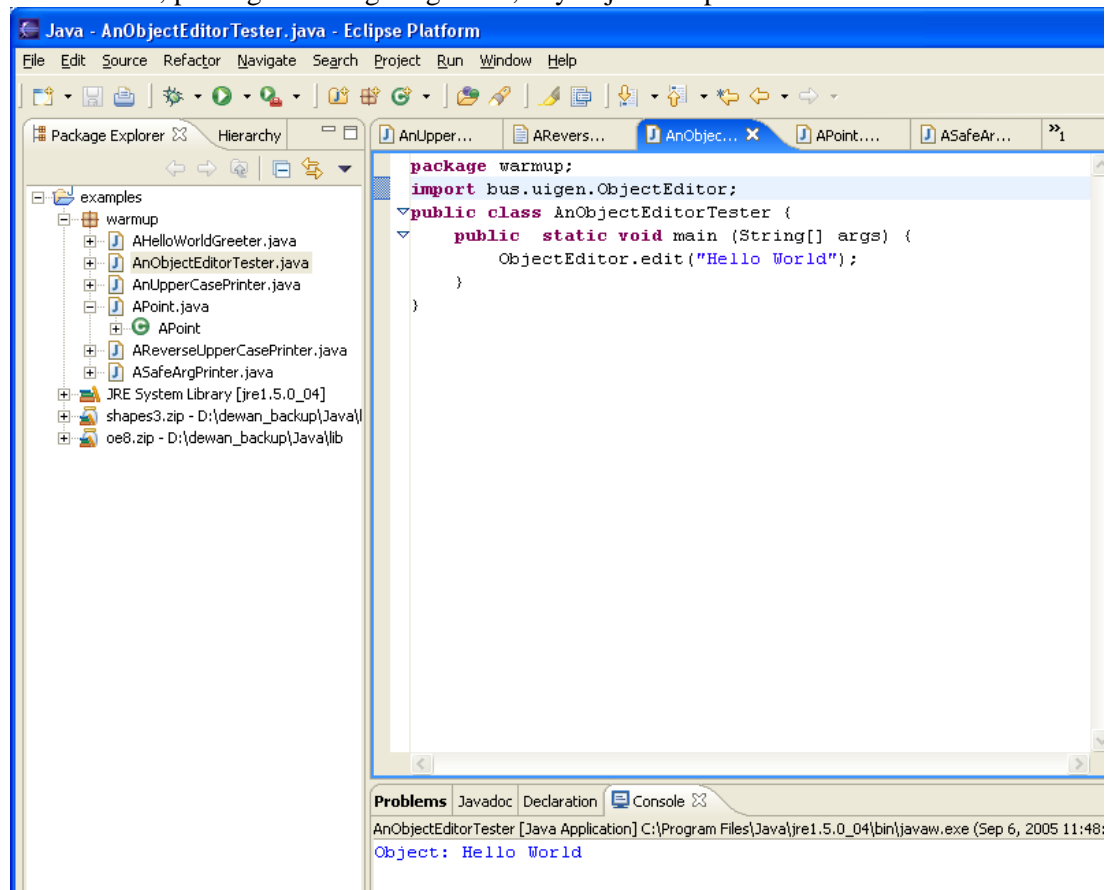
Eclipse adds them to the built-in Java library called JRE:



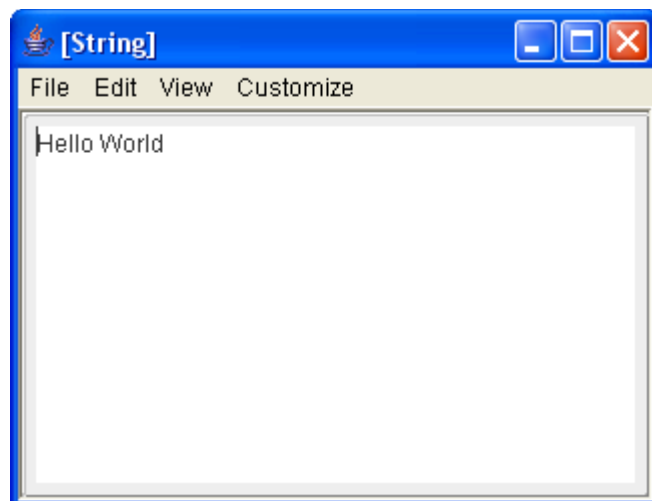
To check the libraries work, let us create a test program, AnObjectEditor tester:



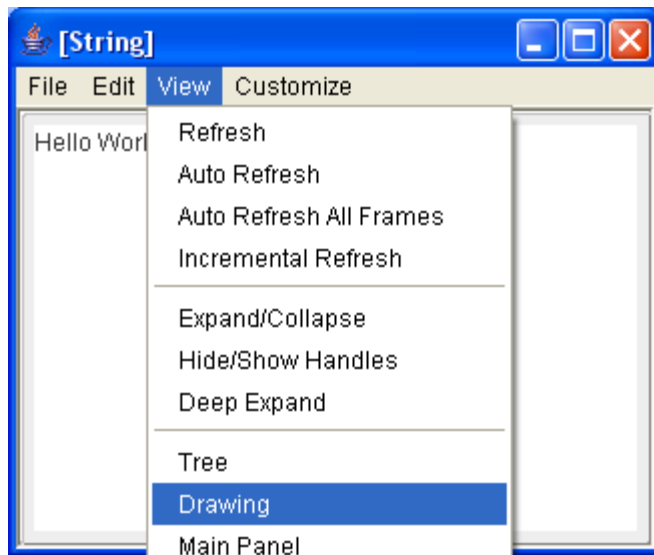
The program imports a particular class in the library, `bus.uigen.ObjectEditor`, and then invokes the `edit()` method on it, passing it a string. In general, any object can be passed to the method.



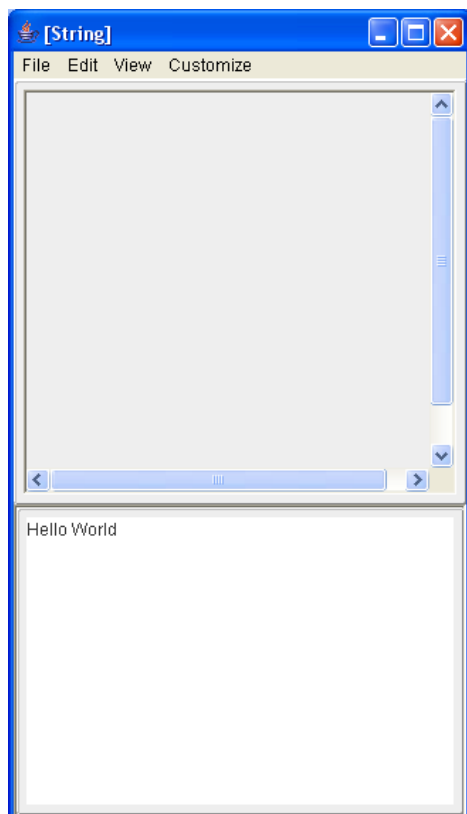
The following ObjectEditor window is created:



At this point we know the `oe` library got linked correctly. To ensure that the `shapes` library, which is the drawing library referenced by `oe`, has also been linked, let us execute the `View→Drawing` command:



An additional drawing window gets created. It is currently empty – that is OK, we have not added any graphics objects to it. If we did not get the window or an exception is thrown by OE, we know the shapes library did not get added.



As mentioned above, you must add these libraries to each project you create.