

Introduction to Programming I

Version 1.0
May 2005

Author

Florence Tiu Balagtas

Team

Joyce Avestro
Florence Balagtas
Rommel Feria
Reginald Hutcherson
Rebecca Ong
John Paul Petines
Sang Shin
Raghavan Srinivas
Matthew Thompson

Requirements For the Laboratory Exercises**Minimum Hardware Configuration**

- **Microsoft Windows operating systems:**
 - **Processor:** 500 MHz Intel Pentium III workstation or equivalent
 - **Memory:** 384 megabytes
 - **Disk space:** 125 megabytes of free disk space
- **SolarisTM operating system:**
 - **Processor:** 450 MHz UltraTM 10 workstation or equivalent
 - **Memory:** 384 megabytes
 - **Disk space:** 125 megabytes of free disk space
- **Linux operating system:**
 - **Processor:** 500 MHz Intel Pentium III workstation or equivalent
 - **Memory:** 384 megabytes
 - **Disk space:** 125 megabytes of free disk space

Recommended Hardware Configuration

- **Microsoft Windows operating systems:**
 - **Processor:** 780 MHz Intel Pentium III workstation or equivalent
 - **Memory:** 512 megabytes
 - **Disk space:** 125 megabytes of free disk space
- **SolarisTM operating system:**
 - **Processor:** 500 MHz UltraTM 60 workstation or equivalent
 - **Memory:** 512 megabytes
 - **Disk space:** 125 megabytes of free disk space
- **Linux operating system:**
 - **Processor:** 800 MHz Intel Pentium III workstation or equivalent
 - **Memory:** 512 megabytes
 - **Disk space:** 125 megabytes of free disk space

Operating System

NetBeans IDE runs on operating systems that support the JavaTM VM. Below is a list of platforms that NetBeans IDE has been tested on.

- Microsoft Windows XP Professional SP1
- Microsoft Windows 2000 Professional SP3
- Solaris operating system (SPARC® Platform Edition), versions 8, 9, and 10
- Solaris operating system (x86 Platform Edition), versions 8, 9, and 10
- Red Hat Linux 9.0
- Red Hat Enterprise Linux 3
- Sun Java Desktop System

NetBeans IDE is also known to run on the following platforms:

- Various other Linux distributions
- Mac OS X 10.1.1 or later
- Open VMS 7.2-1 or later
- Other UNIX® platforms, such as HP-UX

Software

NetBeans IDE runs on the J2SE JDK 5.0 (JavaTM 2 JDK, Standard Edition), which consists of the Java Runtime Environment plus developers tools for compiling, debugging, and running applications written in the JavaTM language. NetBeans IDE 4.0 has also been tested on J2SE SDK version 1.4.2.

For more information, please visit:

<http://www.netbeans.org/community/releases/40/relnotes.html>

Table of Contents

1	Introduction to Computer Programming.....	8
1.1	Introduction.....	8
1.2	Basic Components of a Computer.....	9
1.2.1	Hardware.....	9
1.2.1.1	The Central Processing Unit.....	9
1.2.1.2	Memory	9
1.2.1.3	Input and Output Devices.....	10
1.2.2	Software.....	10
1.3	Overview of Computer Programming Languages.....	11
1.3.1	What is a Programming Language?.....	11
1.3.2	Categories of Programming Languages.....	11
1.4	The Program Development Life Cycle.....	12
1.4.1	Problem Definition.....	13
1.4.2	Problem Analysis.....	13
1.4.3	Algorithm design and representation.....	14
1.4.3.1	Flowcharting Symbols and their meanings.....	15
1.4.4	Coding and Debugging.....	16
1.5	Number Systems and Conversions.....	17
1.5.1	Decimal.....	17
1.5.2	Binary.....	17
1.5.3	Octal.....	17
1.5.4	Hexadecimal.....	17
1.5.5	Conversions.....	18
1.5.5.1	Decimal to Binary / Binary to Decimal.....	18
1.5.5.2	Decimal to Octal (or Hexadecimal) / Octal (or Hexadecimal) to Decimal.....	19
1.5.5.3	Binary to Octal / Octal to Binary.....	20
1.5.5.4	Binary to Hexadecimal / Hexadecimal to Binary.....	21
1.6	Exercises.....	22
1.6.1	Writing Algorithms.....	22
1.6.2	Number Conversions.....	22
2	Introduction to Java.....	23
2.1.1	Java Background.....	23
2.1.1.1	A little Bit of History	23
2.1.1.2	What is Java Technology?.....	23
2.1.2	Phases of a Java Program.....	24
3	Getting to know your Programming Environment.....	26
3.1	My First Java Program.....	26
3.2	Using a Text Editor and Console.....	27
3.2.1	Errors.....	42
3.2.1.1	Syntax Errors.....	42
3.2.1.2	Run-time Errors.....	43
3.3	Using Netbeans.....	44
3.4	Exercises.....	62
3.4.1	Hello World!.....	62
3.4.2	The Tree.....	62
4	Programming Fundamentals.....	63
4.1	Dissecting my first Java program.....	63
4.2	Java Comments.....	65
4.2.1	C++-Style Comments.....	65
4.2.2	C-Style Comments.....	65

4.2.3 Special Javadoc Comments.....	65
4.3 Java Statements and blocks.....	66
4.4 Java Identifiers.....	67
4.5 Java Keywords.....	68
4.6 Java Literals.....	69
4.6.1 Integer Literals	69
4.6.2 Floating-Point Literals	69
4.6.3 Boolean Literals	69
4.6.4 Character Literals	70
4.6.5 String Literals	70
4.7 Primitive data types.....	71
4.7.1 Logical - boolean.....	71
4.7.2 Textual - char.....	71
4.7.3 Integral - byte, short, int & long.....	72
4.7.4 Floating Point - float and double.....	73
4.8 Variables.....	74
4.8.1 Declaring and Initializing Variables.....	74
4.8.2 Outputting Variable Data.....	75
4.8.3 System.out.println() vs. System.out.print()	75
4.8.4 Reference Variables vs. Primitive Variables.....	76
4.9 Operators.....	77
4.9.1 Arithmetic operators.....	77
4.9.1.1 Increment and Decrement operators.....	80
4.9.2 Relational operators.....	82
4.9.3 Logical operators.....	85
4.9.3.1 && (logical AND) and & (boolean logical AND).....	86
4.9.3.2 (logical OR) and (boolean logical inclusive OR).....	88
4.9.3.3 ^ (boolean logical exclusive OR).....	90
4.9.3.4 ! (logical NOT).....	91
4.9.4 Conditional Operator (?:).....	92
4.9.5 Operator Precedence.....	94
4.10 Exercises.....	95
4.10.1 Declaring and printing variables.....	95
4.10.2 Getting the average of three numbers.....	95
4.10.3 Output greatest value.....	95
4.10.4 Operator precedence.....	95
5 Getting Input from the Keyboard.....	96
5.1 Using BufferedReader to get input.....	96
5.2 Using JOptionPane to get input.....	100
5.3 Exercises.....	102
5.3.1 Last 3 words (BufferedReader version).....	102
5.3.2 Last 3 words (JOptionPane version).....	102
6 Control Structures.....	103
6.1 Decision Control Structures.....	103
6.1.1 if statement.....	103
6.1.2 if-else statement.....	105
6.1.3 if-else-if statement.....	107
6.1.4 Common Errors when using the if-else statements:.....	108
6.1.5 Example for if-else-else if.....	109
6.1.6 switch statement.....	110
6.1.7 Example for switch.....	112
6.2 Repetition Control Structures.....	113
6.2.1 while loop.....	113

6.2.2	do-while loop.....	115
6.2.3	for loop.....	116
6.3	Exercises.....	117
6.3.1	Grades.....	117
6.3.2	Number in words.....	117
6.3.3	Hundred Times.....	117
6.3.4	Powers.....	117
7	Java Arrays.....	118
7.1	Introduction to arrays.....	118
7.2	Declaring Arrays.....	119
7.3	Accessing an array element.....	121
7.4	Array length.....	122
7.5	Multidimensional Arrays.....	123
7.6	Exercises.....	124
7.6.1	Days of the Week.....	124
7.6.2	Greatest number.....	124
7.6.3	Addressbook Entries.....	124
8	Command-line Arguments.....	125
8.1	Command-line arguments.....	125
8.2	Command-line arguments in Netbeans.....	127
8.3	Exercises.....	131
8.3.1	Print arguments.....	131
8.3.2	Arithmetic Operations.....	131
9	Working with the Java Class Library.....	132
9.1	Introduction to Object-Oriented Programming.....	132
9.2	Classes and Objects.....	133
9.2.1	Difference Between Classes and Objects.....	133
9.2.2	Class Variables and Methods.....	134
9.2.3	Class Instantiation.....	135
9.3	Methods.....	136
9.3.1	What are Methods and Why Use Methods?.....	136
9.3.2	Calling Instance Methods and Passing Variables.....	137
9.3.3	Passing Variables in Methods.....	138
9.3.3.1	Pass-by-value.....	138
9.3.3.2	Pass-by-reference.....	139
9.3.4	Calling Static Methods.....	140
9.3.5	Scope of a variable.....	141
9.4	Casting, Converting and Comparing Objects.....	145
9.4.1	Casting Primitive Types.....	145
9.4.2	Casting Objects.....	147
9.4.3	Converting Primitive Types to Objects and Vice Versa.....	148
9.4.4	Comparing Objects.....	149
9.4.5	Determining the Class of an Object.....	150
9.5	Exercises.....	151
9.5.1	Defining terms.....	151
9.5.2	Java Scavenger Hunt.....	151
10	Creating your own Classes.....	153
10.1	Defining your own classes.....	153
10.2	Declaring Fields.....	154
10.2.1	Instance Variables.....	154
10.2.2	Class Variables or Static Variables.....	155
10.3	Declaring Methods.....	156
10.3.1.1	get() Methods.....	157

10.3.1.2 Multiple return statements.....	158
10.3.1.3 set() Methods.....	158
10.4 The this reference.....	163
10.5 Overloading Methods.....	164
10.6 Declaring Constructors.....	166
10.7 The this() Constructor Call.....	169
10.8 Packages.....	170
10.8.1 Importing Packages.....	170
10.8.2 Creating your own packages.....	171
10.9 Access Modifiers.....	172
10.9.1 default access (also called package accessibility).....	172
10.9.2 public access.....	173
10.9.3 protected access.....	174
10.9.4 private access.....	175
10.10 Exercises.....	176
10.10.1 Address Book Entry.....	176
10.10.2 AddressBook.....	176
11 Inheritance, Polymorphism and Interfaces.....	177
11.1 Inheritance.....	177
11.1.1 Defining Superclasses and Subclasses.....	178
11.1.2 The super keyword.....	180
11.1.3 Overriding Methods.....	181
11.1.4 Final Methods and Final Classes.....	182
11.2 Polymorphism.....	183
11.3 Abstract Classes.....	186
11.4 Interfaces.....	188
11.5 Exercises.....	191
11.5.1 Extending StudentRecord.....	191
11.5.2 The Shape abstract class.....	191
12 Basic Exception Handling.....	192
12.1 What are Exceptions?.....	192
12.2 Handling Exceptions.....	193
12.3 Exercises.....	195
12.3.1 Catching Exceptions1.....	195
12.3.2 Catching Exceptions 2.....	195
Appendix A : Java and Netbeans Installation.....	196
Installing Java in Linux.....	197
Installing Java in Windows.....	205
Installing Netbeans in Linux.....	207
Installing Netbeans in Windows.....	216
Appendix B: Getting to know your Programming Environment (Windows XP version)..	220
My First Java Program.....	220
Using a Text Editor and Console.....	221
Setting the Path.....	234
Using Netbeans.....	235
Appendix C : Answers to Exercises.....	245
Chapter 1 Exercises.....	245
1.1 Writing Algorithms.....	245
1.2 Number Conversions.....	248
Chapter 2 (No exercises).....	251
Chapter 3 Exercises.....	252
3.1 Hello World!.....	252
3.2 The Tree.....	252

Chapter 4 Exercises.....	253
4.1 Declaring and printing variables.....	253
4.2 Getting the average of three numbers.....	253
4.3 Output greatest value.....	254
4.4 Operator precedence.....	254
Chapter 5 Exercises.....	255
5.1 Last 3 words (BufferedReader version).....	255
5.2 Last 3 words (JOptionPane version).....	256
Chapter 6 Exercises.....	257
6.1 Grades	257
6.2 Number in words.....	259
6.3 Hundred Times.....	261
6.4 Powers.....	264
Chapter 7 Exercises.....	267
7.1 Days of the Week.....	267
7.2 Greatest number.....	268
Chapter 8 Exercises.....	269
8.1 Print Arguments.....	269
Chapter 9 Exercises.....	270
9.1 Defining terms.....	270
9.2 Java Scavenger Hunt.....	270
Chapter 10 Exercises.....	271
10.1 Address Book Entry.....	271
10.2 AddressBook.....	273
Chapter 11 Exercises.....	278
11.1 Extending StudentRecord.....	278
11.2 Abstract Classes.....	281
Chapter 12 Exercises.....	283
12.1 Catching Exceptions 1.....	283
12.2 Catching Exceptions 2.....	283
Appendix D : Machine Problems.....	286
Machine Problem 1: Phone Book.....	286
Machine Problem 2: Minesweeper.....	287
Machine Problem 3: Number Conversion.....	288

1 Introduction to Computer Programming

In this section, we will be discussing the basic components of a computer, both hardware and software. We will also be giving a brief overview of programming languages and the program development life cycle. Finally, different number systems and conversions from one type to another will be discussed.

1.1 Introduction

A computer is a machine that performs a variety of tasks according to specific instructions. It is a data processing machine which accepts data via an ***input device*** and its ***processor*** manipulates the data according to a ***program***.

The computer has two major components. The first one is the **Hardware** which is the tangible part of the computer. It is composed of electronic and mechanical parts.

The second major component is the **software** which is the intangible part of a computer. It consists of data and the computer programs.

1.2 Basic Components of a Computer

1.2.1 Hardware

1.2.1.1 The Central Processing Unit

The processor is the “brain” of the computer. It contains millions of extremely tiny electrical parts. It does the fundamental computing within the system. Examples of processors are Pentium, Athlon and SPARC.

1.2.1.2 Memory

The memory is where data and instructions needed by the CPU to do its appointed tasks can be found. It is divided into several storage locations which have corresponding addresses. The CPU accesses the memory with the use of these addresses.

1. Main Memory

The main memory is very closely connected to the processor. It is used to hold programs and data, that the processor is actively working with. It is not used for long-term storage. It is sometimes called the RAM (Random Access Memory).

The computer's main memory is considered as **volatile** storage. This means that once the computer is turned off, all information residing in the main memory is erased.

2. The Secondary Memory

The secondary memory is connected to main memory. It is used to hold programs and data for long term use. Examples of secondary memory are hard disks and cd-rom.

Secondary memory is considered as **non-volatile** storage. This means that information residing in secondary memory is not erased after the computer is turned off.

Main Memory	Secondary Memory	Property
Fast	Slow	Speed
Expensive	Cheap	Price
Low	High	Capacity
Yes	No	Volatile

Table 1: Comparison between main memory and secondary memory

1.2.1.3 Input and Output Devices

Input and output devices allows a computer system to interact with the outside world by moving data into and out of the system.

Examples of input devices are keyboards, mice and microphones. Examples of output devices are monitors, printers and speakers.

1.2.2 Software

A software is the program that a computer uses in order to function. It is kept on some hardware device like a hard disk, but it itself is intangible. The data that the computer uses can be anything that a program needs. Programs acts like instructions for the processor.

Some Types of Computer Programs:

1. Systems Programs

- Programs that are needed to keep all the hardware and software systems running together smoothly
- Examples:
 - Operating Systems like Linux, Windows, Unix, Solaris, MacOS

2. Application Programs

- Programs that people use to get their work done
- Examples:
 - Word Processor
 - Game programs
 - Spreadsheets

3. Compilers

- The computer understands only one language: machine language. Machine language is in the form of ones and zeros. Since it is highly impractical for people to create programs out of zeros and ones, there must be a way of translating or converting a language which we understand into machine language, for this purpose, there exists compilers.

1.3 Overview of Computer Programming Languages

1.3.1 What is a Programming Language?

A programming language is a standardized communication technique for expressing instructions to a computer. Like human languages, each language has its own syntax and grammar.

Programming languages enable a programmer to precisely specify what data a computer will act upon, how these data will be stored/transmitted, and precisely what actions to take under various circumstances.

There are different types of programming languages that can be used to create programs, but regardless of what language you use, these instructions are translated into machine language that can be understood by computers.

1.3.2 Categories of Programming Languages

1. High-level Programming Languages

- A high-level programming language is a programming language that is more user-friendly, to some extent platform-independent, and abstract from low-level computer processor operations such as memory accesses. A programming statement may be translated into one or several machine instructions by a **compiler**.
- Examples are Java, C, C++, Basic, Fortran

2. Low-level Assembly Language

- Assembly languages are similar to machine languages, but they are much easier to program in because they allow a programmer to substitute names for numbers. Assembly languages are available for each CPU family, and each assembly instruction is translated into one machine instruction by an **assembler** program.

Note: The terms "high-level" and "low-level" are inherently relative. Originally, assembly language was considered low-level and COBOL, C, etc. were considered high-level. Many programmers today might refer to these latter languages as low-level.

1.4 The Program Development Life Cycle

Programmers do not sit down and start writing code right away when trying to make a computer program. Instead, they follow an organized plan or methodology, that breaks the process into a series of tasks.

Here are the basic steps in trying to solve a problem on the computer:

1. Problem Definition
2. Problem Analysis
3. Algorithm design and representation (Pseudocode or flowchart)
4. Coding and debugging

In order to understand the basic steps in solving a problem on a computer, let us define a single problem that we will solve step-by-step as we discuss the problem solving methodologies in detail. The problem we will solve will be defined in the next section.

1.4.1 Problem Definition

A programmer is usually given a task in the form of a problem. Before a program can be designed to solve a particular problem, the problem must be well and clearly defined first in terms of its input and output requirements.

A clearly defined problem is already half the solution. Computer programming requires us to define the problem first before we even try to create a solution.

Let us now define our example problem:

“Create a program that will determine the number of times a name occurs in a list.”

1.4.2 Problem Analysis

After the problem has been adequately defined, the simplest and yet the most efficient and effective approach to solve the problem must be formulated.

Usually, this step involves breaking up the problem into smaller and simpler sub-problems.

Example Problem:

Determine the number of times a name occurs in a list

Input to the program:

list of names, name to look for

Output of the program:

the number of times the name occurs in a list

1.4.3 Algorithm design and representation

Once our problem is clearly defined, we can now set to finding a solution. In computer programming, it is normally required to express our solution in a step-by-step manner.

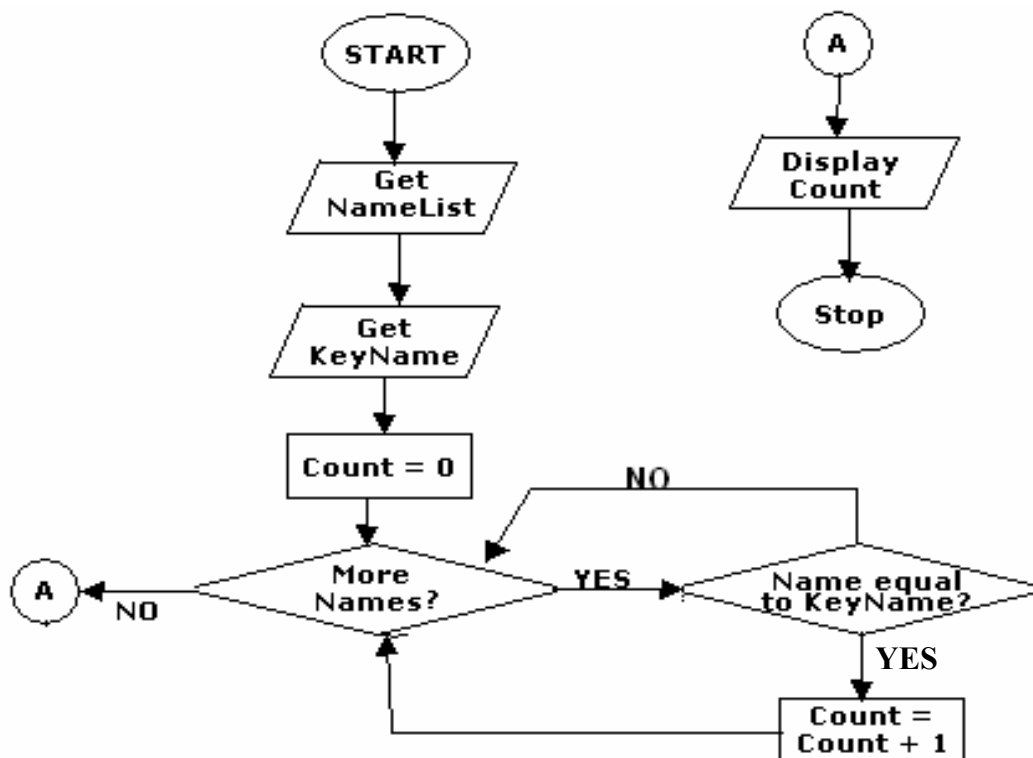
An **Algorithm** is a clear and unambiguous specification of the steps needed to solve a problem. It may be expressed in either **Human language** (English, Tagalog), through a graphical representation like a **flowchart** or through a **pseudocode**, which is a cross between human language and a programming language.

Now given the problem defined in the previous sections, how do we express our general solution in such a way that it is simple yet understandable?

Expressing our solution through Human language:

1. Get the list of names
2. Get the name to look for, let's call this the keyname
3. Compare the keyname to each of the names in the list
4. If the keyname is the same with a name in the list, add 1 to the count
5. If all the names have been compared, output the result

Expressing our solution through a flowchart:



Expressing our solution through pseudocode:

```



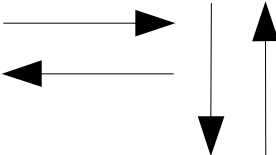
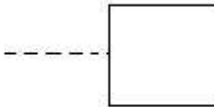
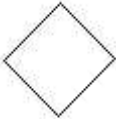

Let nameList = List of Names
Let keyName = the name to be sought
Let Count = 0
For each name in NameList do the following
  if name == keyName
    Count = Count + 1
Display Count

```

1.4.3.1 Flowcharting Symbols and their meanings

A flowchart is a design tool used to graphically represent the logic in a solution. Flowcharts typically do not display programming language commands. Rather, they state the concept in English or mathematical notation.

Here are some guidelines for commonly used symbols in creating flowcharts. You can use any symbols in creating your flowcharts, as long as you are consistent in using them.

Symbol	Name	Meaning
	Process Symbol	Represents the process of executing a defined operation or groups of operations that results in a change in value, form, or location of information. Also functions as the default symbol when no other symbol is available.
	Input/Output (I/O) Symbol	Represents an I/O function, which makes data available for processing (input) or displaying (output) of processed information.
	Flowline Symbol	Represents the sequence of available information and executable operations. The lines connect other symbols, and the arrowheads are mandatory only for right-to-left and bottom-to-top flow.
	Annotation Symbol	Represents the addition of descriptive information, comments, or explanatory notes as clarification. The vertical line and the broken line may be placed on the left, as shown, or on the right.
	Decision Symbol	Represents a decision that determines which of a number of alternative paths is to be followed.
	Terminal Symbol	Represents the beginning, the end, or a point of interruption or delay in a program.



<i>Symbol</i>	<i>Name</i>	<i>Meaning</i>
	Connector Symbol	Represents any entry from, or exit to, another part of the flowchart. Also serves as an off-page connector.
	Predefined Process Symbol	Represents a named process consisting of one or more operations or program steps that are specified elsewhere.

Table 2: Flowchart Symbols

1.4.4 Coding and Debugging

After constructing the algorithm, it is now possible to create the source code. Using the algorithm as basis, the source code can now be written using the chosen programming language.

Most of the time, after the programmer has written the program, the program isn't 100% working right away. The programmer has to add some fixes to the program in case of errors (also called bugs) that occurs in the program. This process of is called **debugging**.

There are two types of errors that a programmer will encounter along the way. The first one is compile-time error, and the other is runtime error.

Compile-Time Errors occur if there is a syntax error in the code. The compiler will detect the error and the program won't even compile. At this point, the programmer is unable to form an executable that a user can run until the error is fixed.

Forgetting a semi-colon at the end of a statement or misspelling a certain command, for example, is a compile-time error. It's something the compiler can detect as an error.

Compilers aren't perfect and so can't catch all errors at compile time. This is especially true for logic errors such as infinite loops. This type of error is called **runtime error**.

For example, the actual syntax of the code looks okay. But when you follow the code's logic, the same piece of code keeps executing over and over again infinitely so that it loops. In such a case, compilers aren't really smart enough to catch all of these types of errors at compile-time, and therefore, the program compiles fine into an executable file. However, and unfortunately, when the end-user runs the program, the program (or even their whole computer) freezes up due to an infinite loop. Other types of run-time errors are when an incorrect value is computed, the wrong thing happens, etc.

1.5 Number Systems and Conversions

Numbers can be represented in a variety of ways. The representation depends on what is called the **BASE**. The following are the four most common representations.

1.5.1 Decimal

We normally represent numbers in their decimal form. Numbers in decimal form are in base 10. This means that the only digits that appear are 0-9. Here are examples of numbers written in decimal form:

126_{10} (normally written as just 126)
 11_{10} (normally written as just 11)

1.5.2 Binary

Numbers in binary form are in base 2. This means that the only legal digits are 0 and 1. We need to write the subscript ₂ to indicate that the number is a binary number. Here are examples of numbers written in binary form:

1111110_2
 1011_2

1.5.3 Octal

Numbers in octal form are in base 8. This means that the only legal digits are 0-7. We need to write the subscript ₈ to indicate that the number is an octal number. Here are examples of numbers written in octal form:

176_8
 13_8

1.5.4 Hexadecimal

Numbers in hexadecimal form are in base 16. This means that the only legal digits are 0-9 and the letters A-F (or a-f, lowercase or uppercase does not matter). We need to write the subscript ₁₆ to indicate that the number is a hexadecimal number. Here are examples of numbers written in hexadecimal form:

$7E_{16}$
 B_{16}

Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Decimal Equivalent	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Table 3: Hexadecimal Numbers and their Equivalence to decimal numbers

Decimal	Binary	Octal	Hexadecimal
126 ₁₀	1111110 ₂	176 ₈	7E ₁₆
11 ₁₀	1011 ₂	13 ₈	B ₁₆

Table 4: Summary of Examples

1.5.5 Conversions

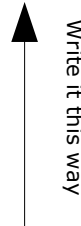
1.5.5.1 Decimal to Binary / Binary to Decimal

To convert a decimal number to binary, continuously divide the number by 2 and get the remainder (which is either 0 or 1), and get that number as a digit of the binary form of the number. Get the quotient and divide that number again by 2 and repeat the whole process until the quotient reaches 0 or 1. We then get all the remainders starting from the last remainder, and the result is the binary form of the number.

NOTE: For the last digit which is already less than the divisor (which is 2) just copy the value to the remainder portion.

For Example:

$$126_{10} = ?_2$$

	Quotient	Remainder	
126 / 2 =	63	0	 Write it this way
63 / 2 =	31	1	
31 / 2 =	15	1	
15 / 2 =	7	1	
7 / 2 =	3	1	
3 / 2 =	1	1	
1 / 2 =		1	

So, writing the remainders from the bottom up, we get the binary number 1111110₂

To convert a binary number to decimal, we multiply the binary digit to "2 raised to the position of the binary number". We then add all the products to get the resulting decimal number.

For Example:

$$1111110_2 = ?_{10}$$

Position	6	5	4	3	2	1	0	
Binary Digits	1	1	1	1	1	1	0	
								0 x 2 ⁰ = 0
								1 x 2 ¹ = 2
								1 x 2 ² = 4
								1 x 2 ³ = 8
								1 x 2 ⁴ = 16
								1 x 2 ⁵ = 32
								1 x 2 ⁶ = 64
								TOTAL: 126

1.5.5.2 Decimal to Octal (or Hexadecimal) / Octal (or Hexadecimal) to Decimal

Converting decimal numbers to Octal or hexadecimal is basically the same as converting decimal to binary. However, instead of having 2 as the divisor, you replace it with 8 (for octal) or 16 (for hexadecimal).

For Example (Octal):

$$126_{10} = \underline{\quad}_8$$

	Quotient	Remainder
126 / 8 =	15	6
15 / 8 =	1	7
1 / 8 =	0	1

Write it this way
↑

So, writing the remainders from the bottom up, we get the octal number 176_8

For Example (Hexadecimal):

$$126_{10} = \underline{\quad}_{16}$$

	Quotient	Remainder
126 / 16 =	7	14 (equal to hex digit E)
7 / 16 =	0	7

Write it this way
↑

So, writing the remainders from the bottom up, we get the hexadecimal number $7E_{16}$
* * *

Converting octal or hexadecimal numbers is also the same as converting binary numbers to decimal. To do that, we will just replace the base number 2 with 8 for Octal and 16 for hexadecimal.

For Example (Octal):

$$176_8 = \underline{\quad}_{10}$$

Position	2	1	0	
Octal Digits	1	7	6	
				$6 \times 8^0 = 6$
				$7 \times 8^1 = 56$
				$1 \times 8^2 = 64$
				TOTAL: 126

For Example (Hexadecimal):

$$7E_{16} = ?_{10}$$

Position	1	0
Hex Digits	7	E
		$14 \times 16^0 = 14$
		$7 \times 16^1 = 112$
		TOTAL: 126

1.5.5.3 Binary to Octal / Octal to Binary

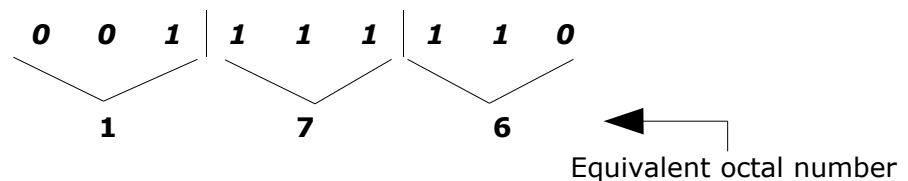
To convert from binary numbers to octal, we partition the binary number into groups of 3 digits (from right to left), and pad it with zeros if the number of digits is not divisible by 3. We then convert each partition into its corresponding octal digit. The following is a table showing the binary representation of each octal digit.

Octal Digit	Binary Representation
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Table 5: Octal Digits and their corresponding binary representation

For Example:

$$1111110_2 = ?_8$$



Converting octal numbers to binary is just the opposite of what is given above. Simply convert each octal digit into its binary representation (given the table) and concatenate them. The result is the binary representation.

1.5.5.4 Binary to Hexadecimal / Hexadecimal to Binary

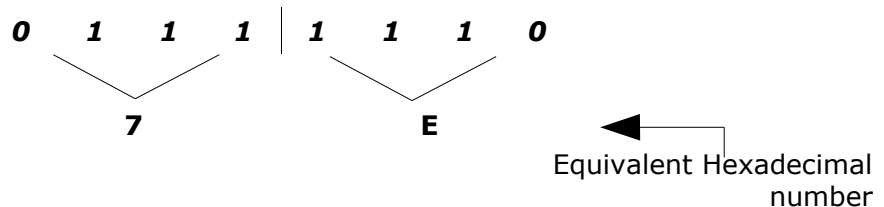
To convert from binary numbers to hexadecimal, we partition the binary number into groups of 4 digits (from right to left), and pad it with zeros if the number of digits is not divisible by 4. We then convert each partition into its corresponding hexadecimal digit. The following is a table showing the binary representation of each hexadecimal digit.

Hexadecimal Digit	Binary Representation
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Table 6: Hexadecimal Digits and their corresponding binary representation

For Example:

$$1111110_2 = ?_{16}$$



Converting hexadecimal numbers to binary is just the opposite of what is given above. Simply convert each hexadecimal digit into its binary representation (given the table) and concatenate them. The result is the binary representation.

1.6 Exercises

1.6.1 Writing Algorithms

Given the following set of tasks, create an algorithm to accomplish the following tasks. You may write your algorithms using pseudocodes or you can use flowcharts.

1. Baking Bread
2. Logging into your laboratory's computer
3. Getting the average of three numbers

1.6.2 Number Conversions

Convert the following numbers:

1. 1980_{10} to binary, hexadecimal and octal
2. 1001001101_2 to decimal, hexadecimal and octal
3. 76_8 to binary, hexadecimal and decimal
4. $43F_{16}$ to binary, decimal and octal

2 Introduction to Java

In this section, we will be discussing a little bit of Java history and what is Java Technology. We will also discuss the phases that a Java program undergoes.

2.1.1 Java Background

2.1.1.1 A little Bit of History

Java was created in 1991 by James Gosling et al. of Sun Microsystems. Initially called Oak, in honor of the tree outside Gosling's window, its name was changed to Java because there was already a language called Oak.

The original motivation for Java was the need for platform independent language that could be embedded in various consumer electronic products like toasters and refrigerators. One of the first projects developed using Java was a personal hand-held remote control named Star 7.

At about the same time, the World Wide Web and the Internet were gaining popularity. Gosling et. al. realized that Java could be used for Internet programming.

2.1.1.2 What is Java Technology?

The Java technology is:

- A programming language
- A development environment
- An application environment
- A deployment environment

As a **programming language**, Java can create all kinds of applications that you could create using any conventional programming language.

As a **development environment**, Java technology provides you with a large suite of tools: a compiler, an interpreter, a documentation generator, a class file packaging tool, and so on.

Java technology applications are typically general-purpose programs that run on any machine where the **Java runtime environment** (JRE) is installed.

There are **two main deployment environments**: **First, the JRE** supplied by the Java 2 Software Development Kit (SDK) contains the complete set of class files for all the Java technology packages, which includes basic language classes, GUI component classes, and so on. The other main deployment environment is on your **web browser**. Most commercial browsers supply a Java technology interpreter and runtime environment.

2.1.2 Phases of a Java Program

The following figure describes the process of compiling and executing a Java program.

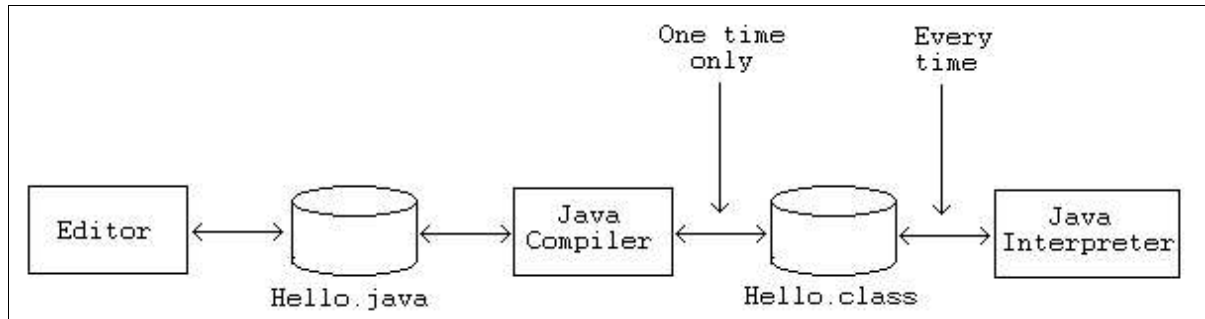


Figure 2.1: Phases of a Java Program

The first step in creating a Java program is by writing your programs in a text editor. Examples of text editors you can use are notepad, vi, emacs, etc. This file is stored in a disk file with the extension `.java`.

After creating and saving your Java program, compile the program by using the Java Compiler. The output of this process is a file of Java **bytecodes** with the file extension `.class`.

A **bytecode** is a special machine language that can be understood by the **Java Virtual Machine (JVM)**. The bytecode is independent of any particular computer hardware, so any computer with a Java interpreter can execute the compiled Java program, no matter what type of computer the program was compiled on.

The **Java Virtual Machine** is an imaginary machine that is implemented by emulating software on a real machine. The JVM provides the hardware platform specifications to which you compile all Java technology code. This specification enables the Java software to be platform-independent because the compilation is done for a generic machine known as the JVM.

The `.class` file is then interpreted by the Java interpreter that converts the bytecodes into the machine language of the particular computer you are using.

<i>Task</i>	<i>Tool to use</i>	<i>Output</i>
Write the program	Any text editor	File with <code>.java</code> extension
Compile the program	Java Compiler	File with <code>.class</code> extension (Java bytecodes)
Run the program	Java Interpreter	Program Output

Table 7: Summary of Phases of a Java Program

3 Getting to know your Programming Environment

In this section, we will be discussing on how to write, compile and run Java programs. There are two ways of doing this, the first one is by using a console and a text editor. The second one is by using Netbeans which is an **Integrated Development Environment or IDE**.

An IDE is a programming environment integrated into a software application that provides a GUI builder, a text or code editor, a compiler and/or interpreter and a debugger.

This tutorial uses RedHat Linux as the operating system. Make sure that before you do this tutorial, you have installed Java and Netbeans in your system. For instructions on how to install Java and Netbeans, please refer to **Appendix A**. For the Windows XP version of this section, please refer to **Appendix B**.

Before going into details, let us first take a look at the first Java program you will be writing.

3.1 *My First Java Program*

```
public class Hello
{
    /**
     * My first java program
     */
    public static void main(String[] args) {

        //prints the string "Hello world" on screen
        System.out.println("Hello world!");

    }
}
```

Before we try to explain what the program means, let's first try to write this program in a file and try to run it.

3.2 Using a Text Editor and Console

For this example, we will be using a text editor to edit the Java source code. You will also need to open the Terminal window to compile and execute your Java programs.

Step 1: Start the Text Editor

To start the Text Editor in Linux, click on Menu-> Accessories-> Text Editor.

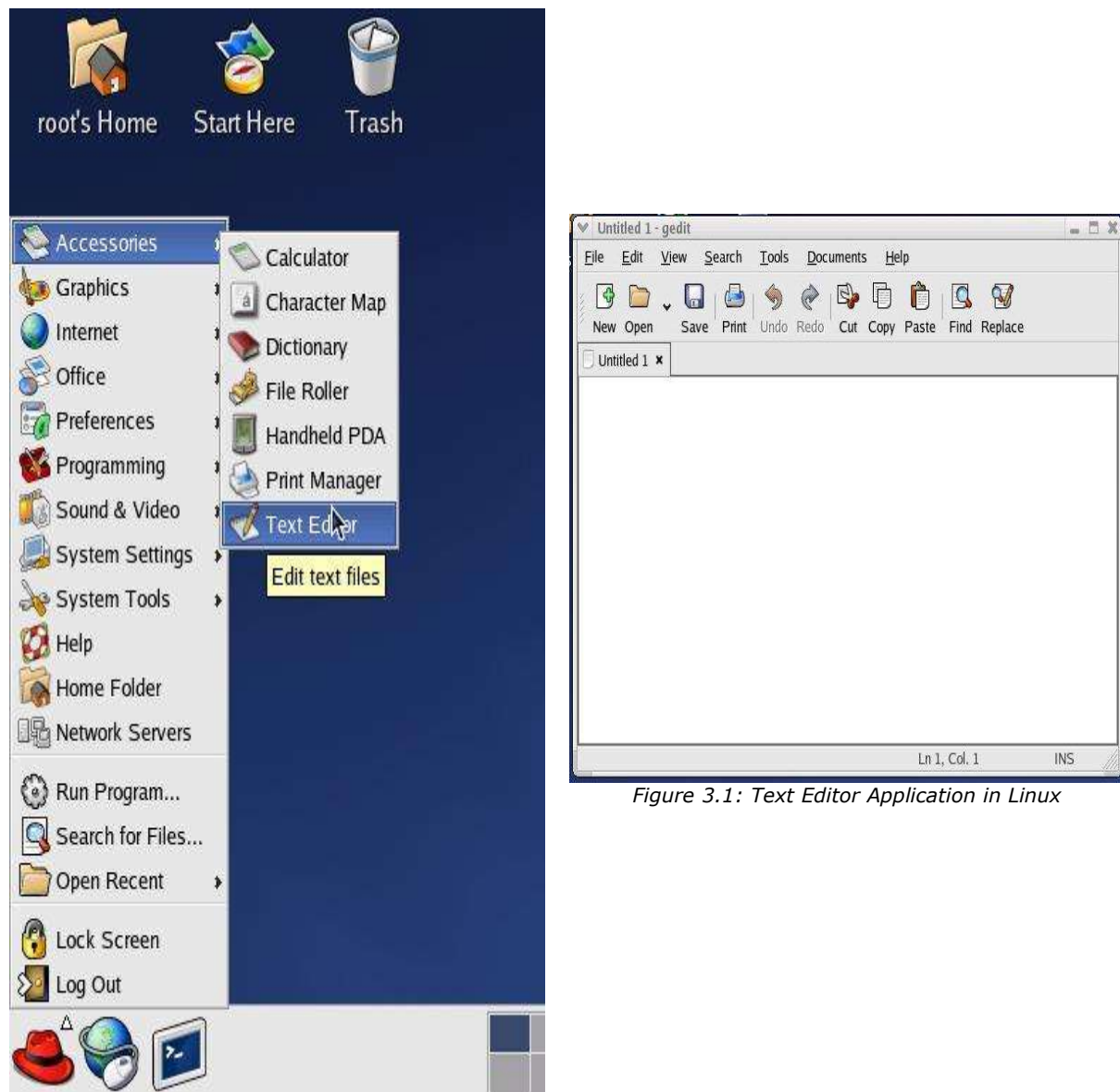


Figure 3.1: Text Editor Application in Linux

Step 2: Open Terminal

To open Terminal in Linux, click on Menu-> System Tools-> Terminal.

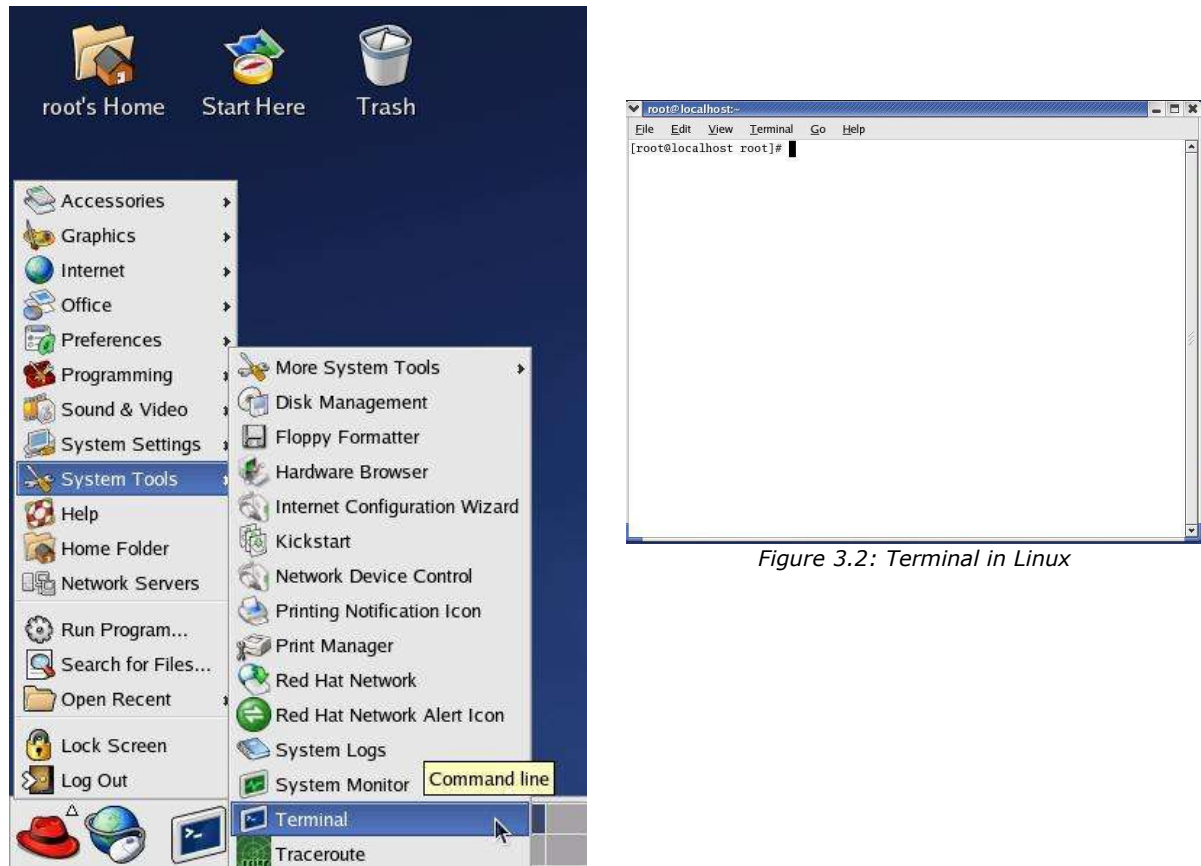
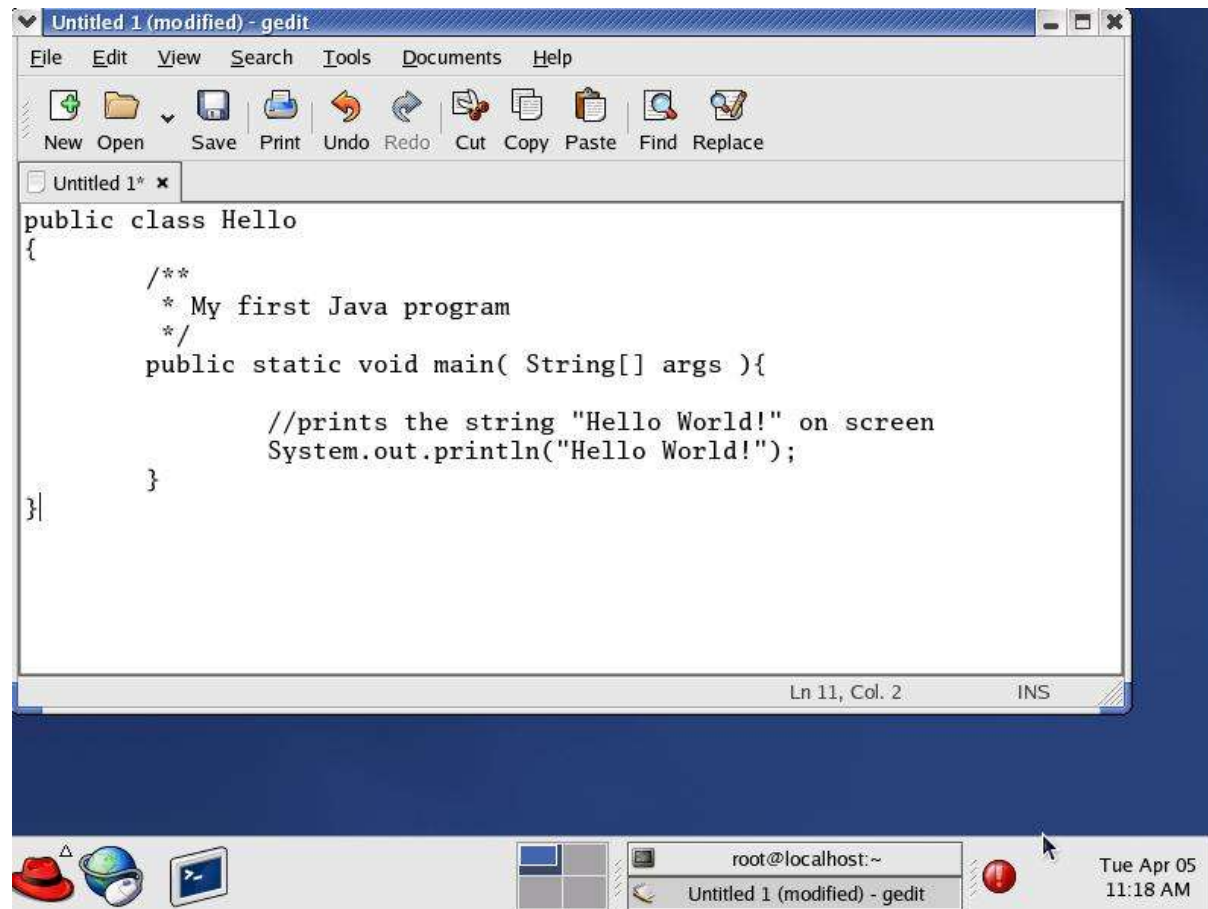


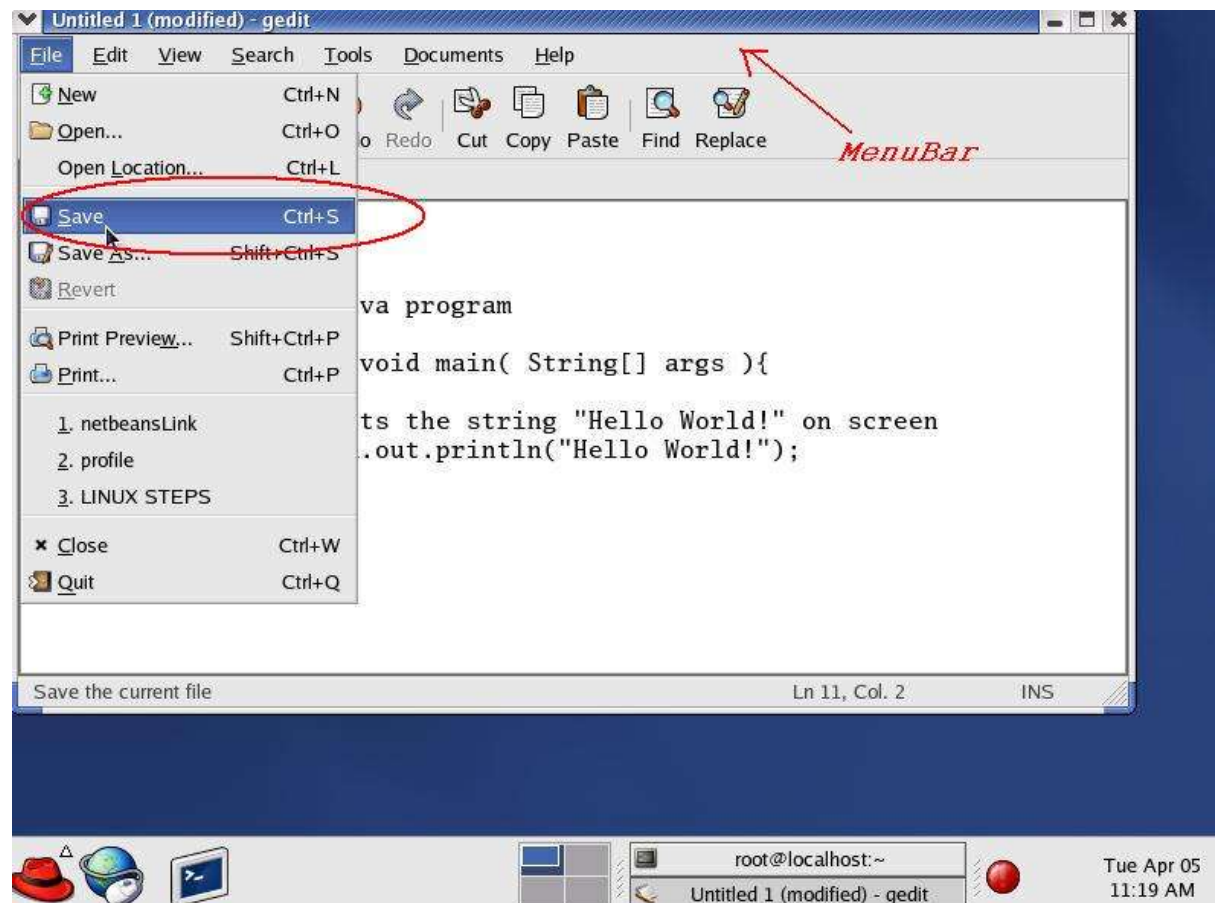
Figure 3.2: Terminal in Linux

Step 3: Write your the source code of your Java program in the Text Editor

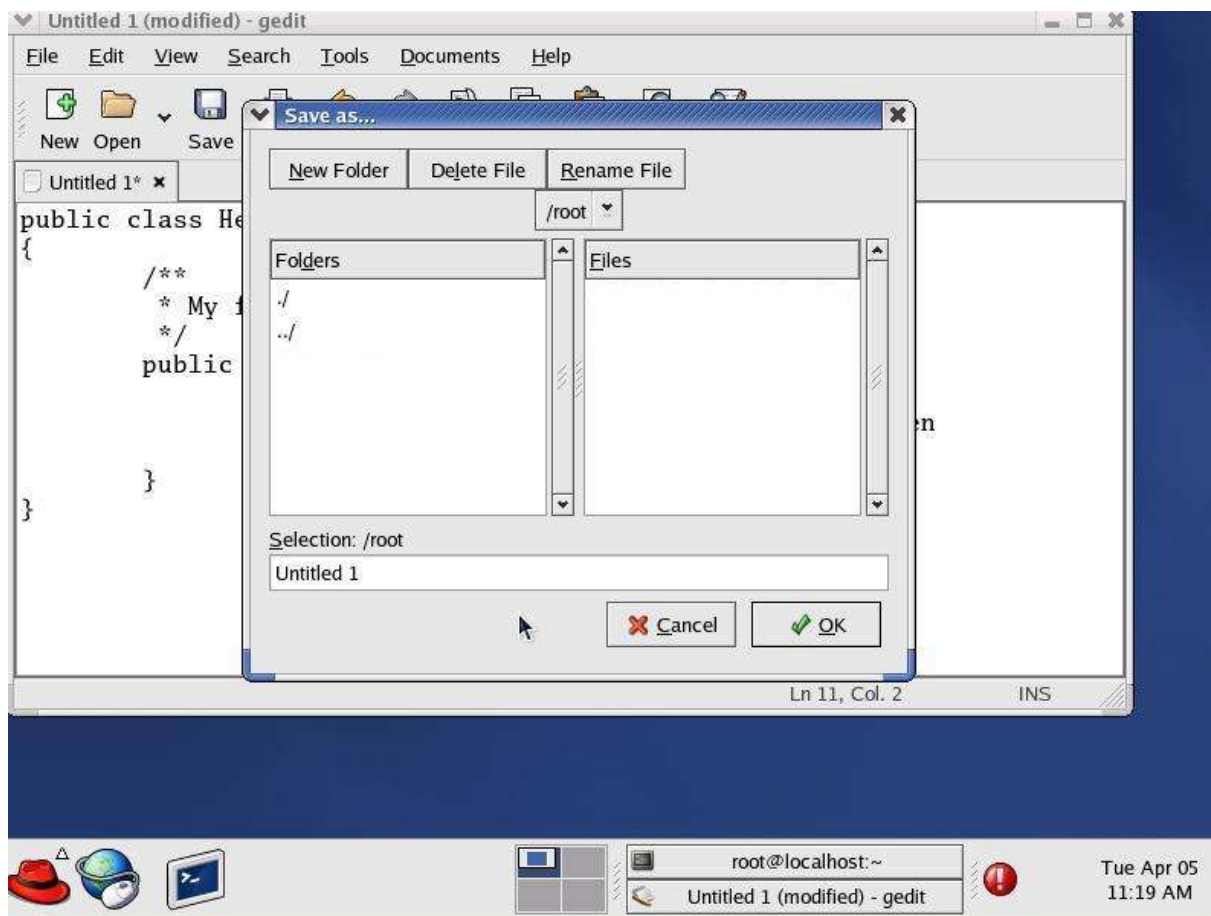
Step 4: Save your Java Program

We will save our program on a file named "Hello.java", and we will be saving it inside a folder named MYJAVAPROGRAMS.

To open the **Save** dialog box, click on the File menu found on the menubar and then click on Save.

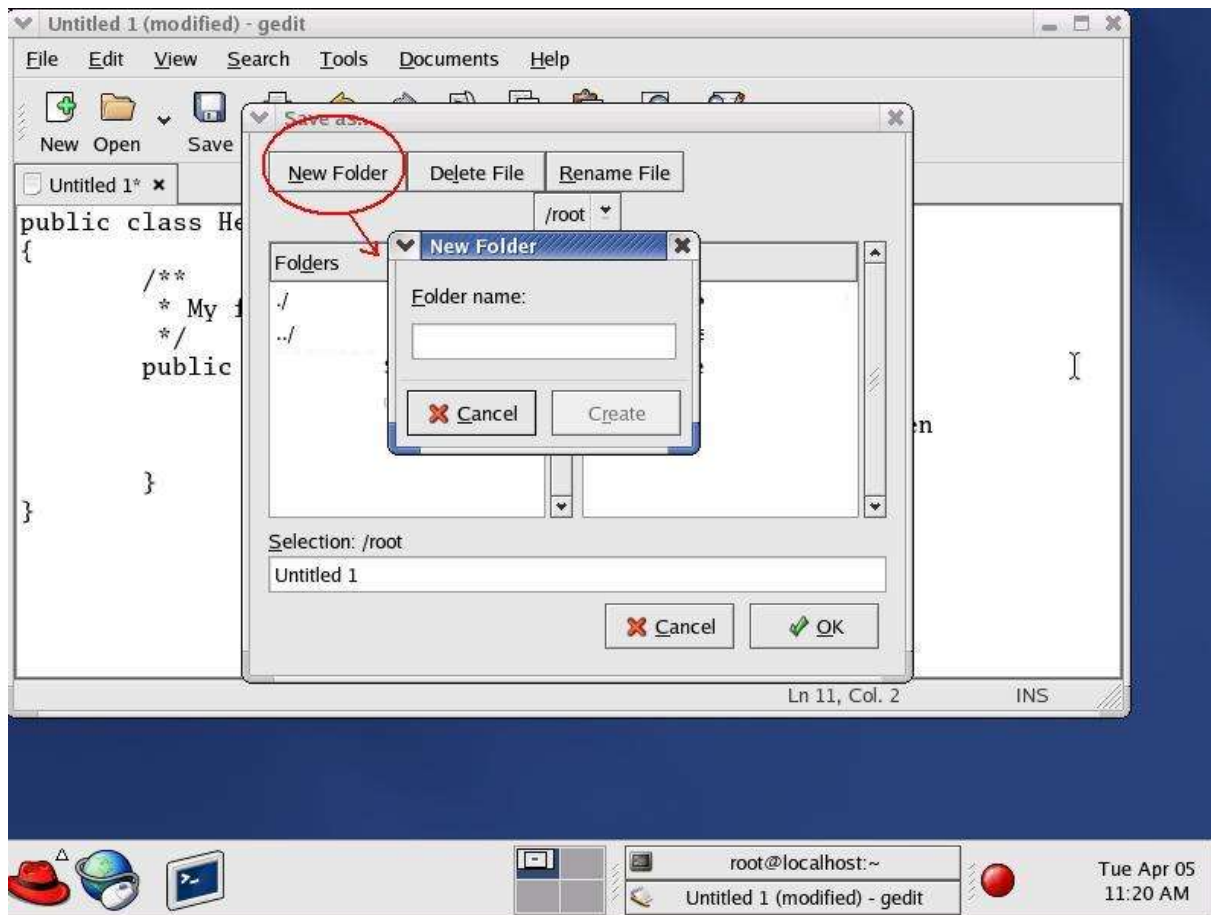


After doing the procedure described above, a dialog box will appear as shown in Figure below.

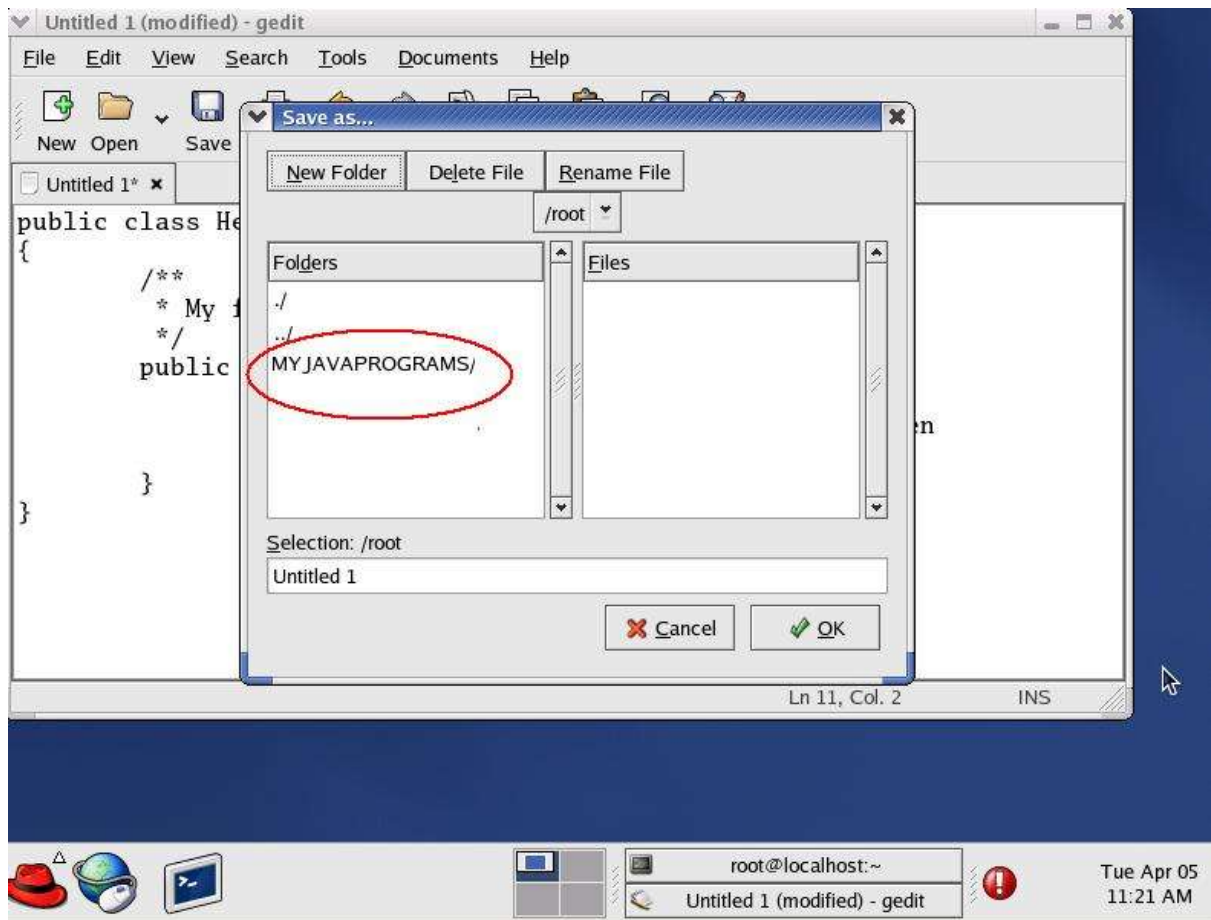


Now, we'll create a new folder inside the root folder where we will save your programs. We shall name this folder MYJAVAPROGRAMS. Click on the button encircled in the figure below to create the folder.

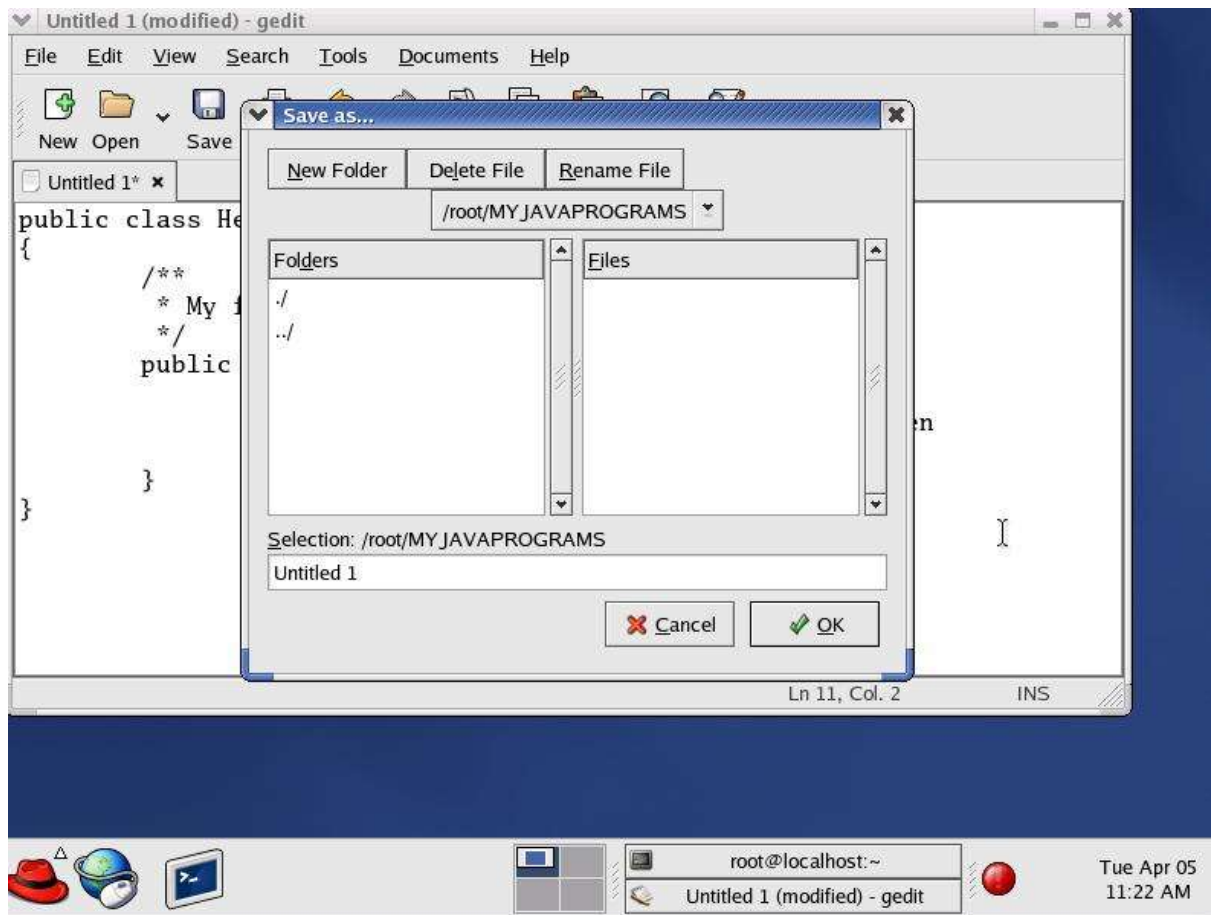
A dialog box named "New Folder" will then appear. Type on the "Folder Name" Textbox MYJAVAPROGRAMS, and click on the CREATE button.



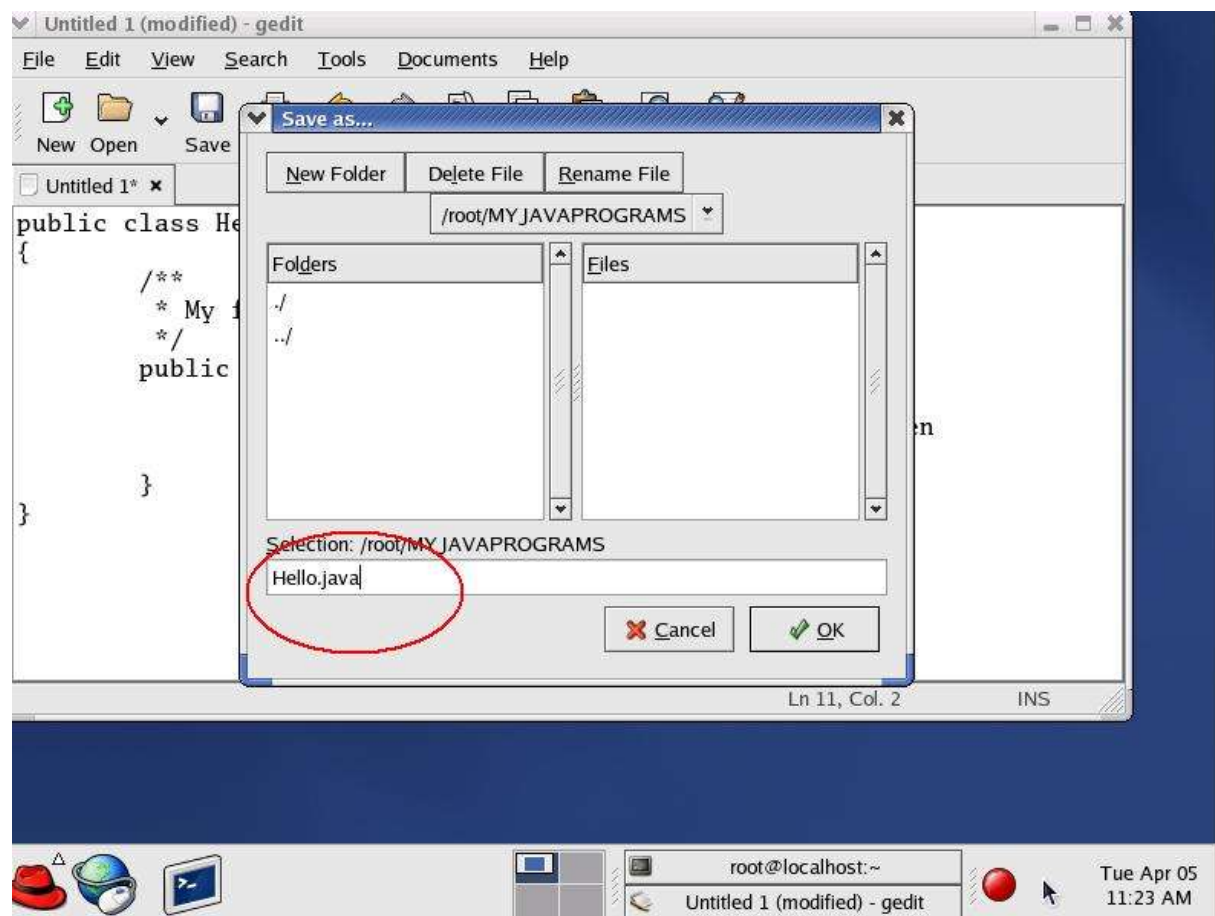
Now that we've created the folder where we will save all the files, double click on that folder to open it.



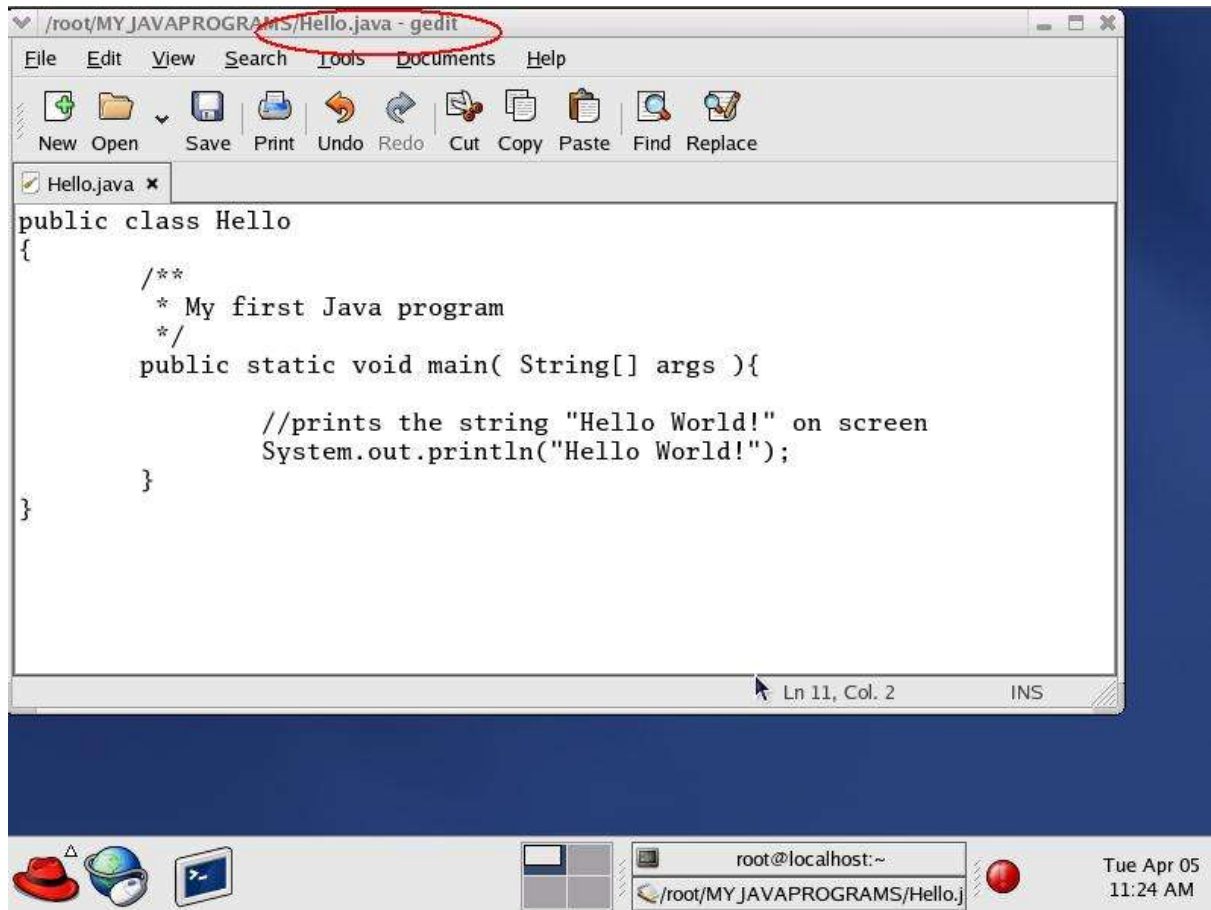
You will see a similar figure as shown below after you clicked on MYJAVAPROGRAMS. The folder should be empty for now since it's a newly created folder and we haven't saved anything in it yet.



Now, in the Selection textbox, type in the filename of your program, which is "Hello.java", and then click on the OK button.



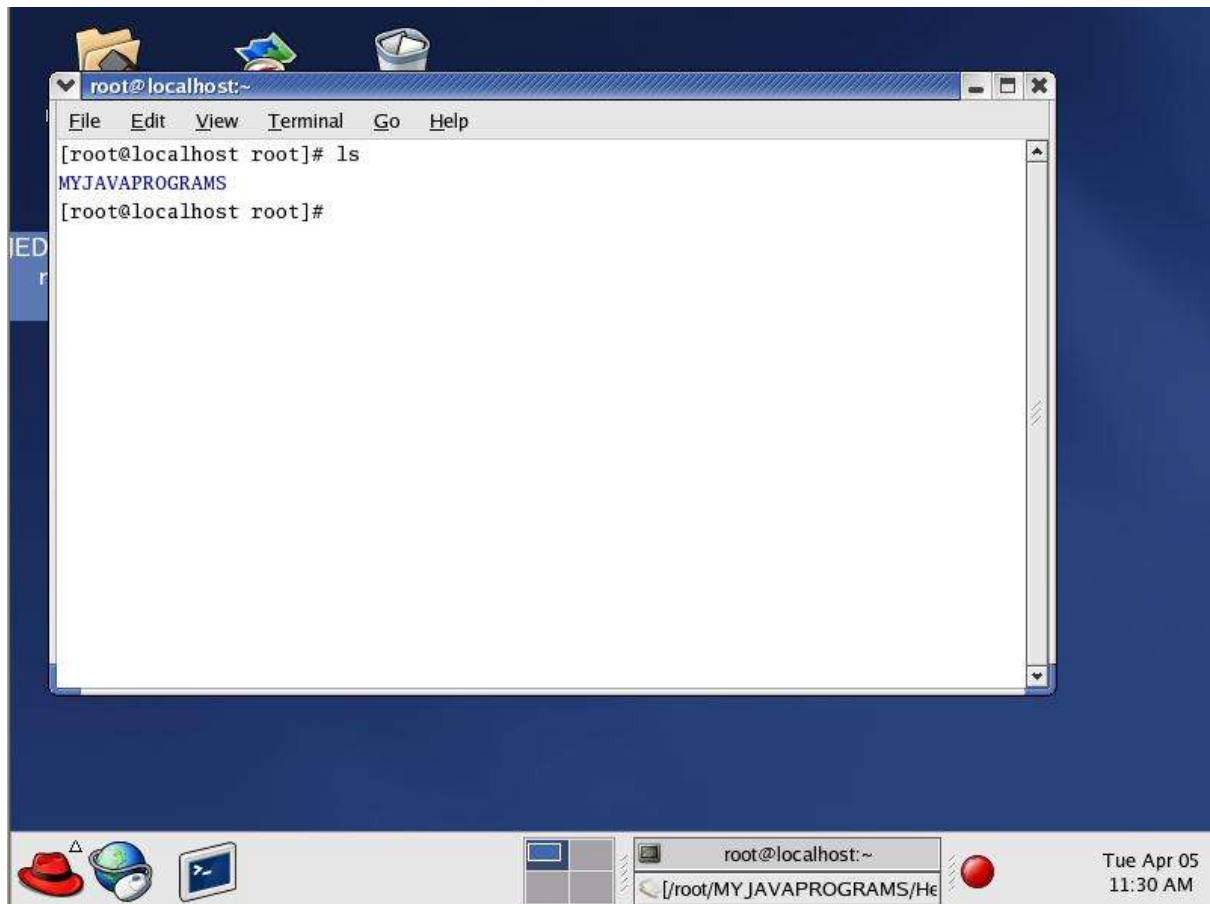
Now that you've saved your file, notice how the title of the frame changes from "Untitled 1 (modified) - gedit" to "/root/MYJAVAPROGRAMS/Hello.java - gedit". Take note that if you want to make changes in your file, you can just edit it, and then save it again by clicking on File -> Save.



Step 5: Compiling your program

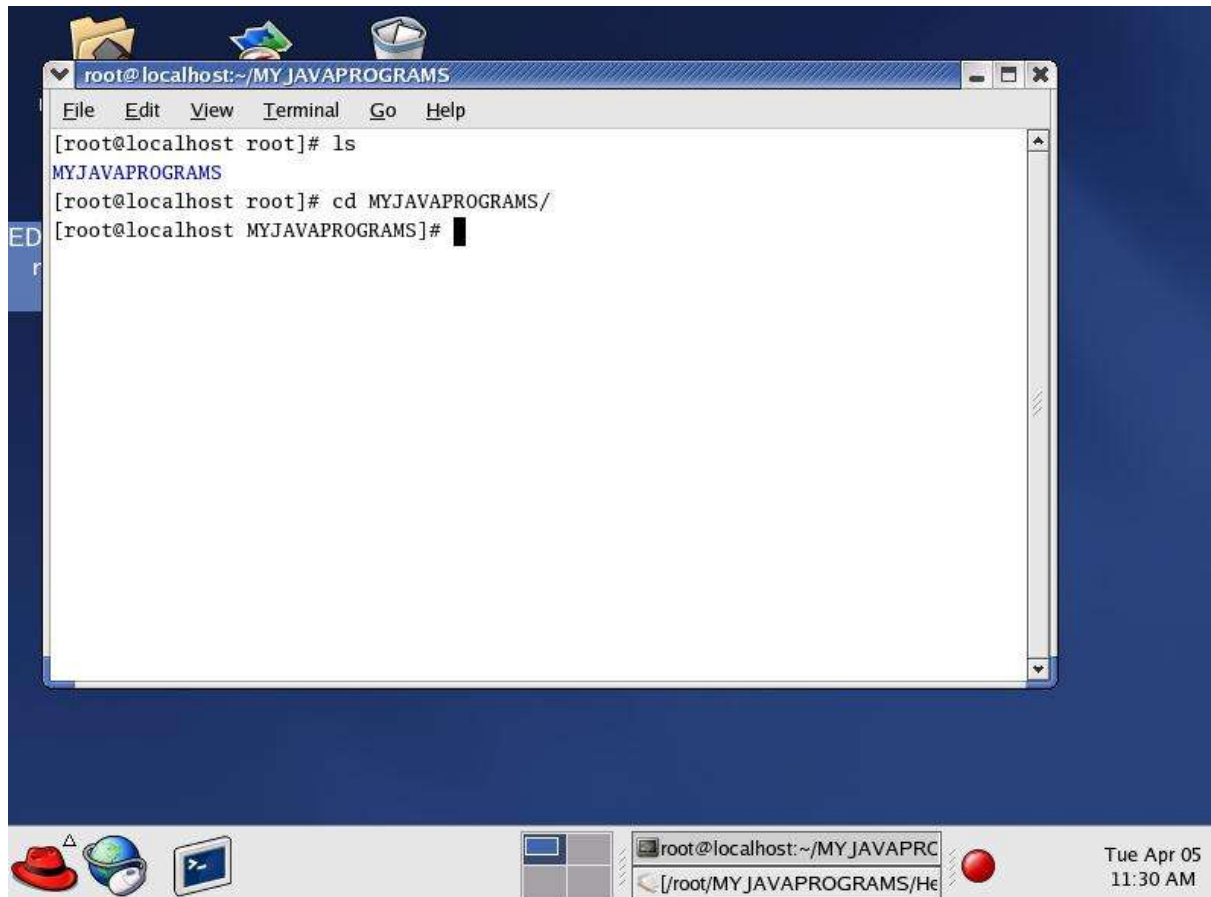
Now, the next step is to compile your program. Go to the Terminal window we just opened a while ago.

Typically, when you open the terminal window, it opens up and takes you directly to what is called your **home folder**. To see what is inside that home folder, type **ls** and then press ENTER. What you will see is a list of files and folders inside your home folder.

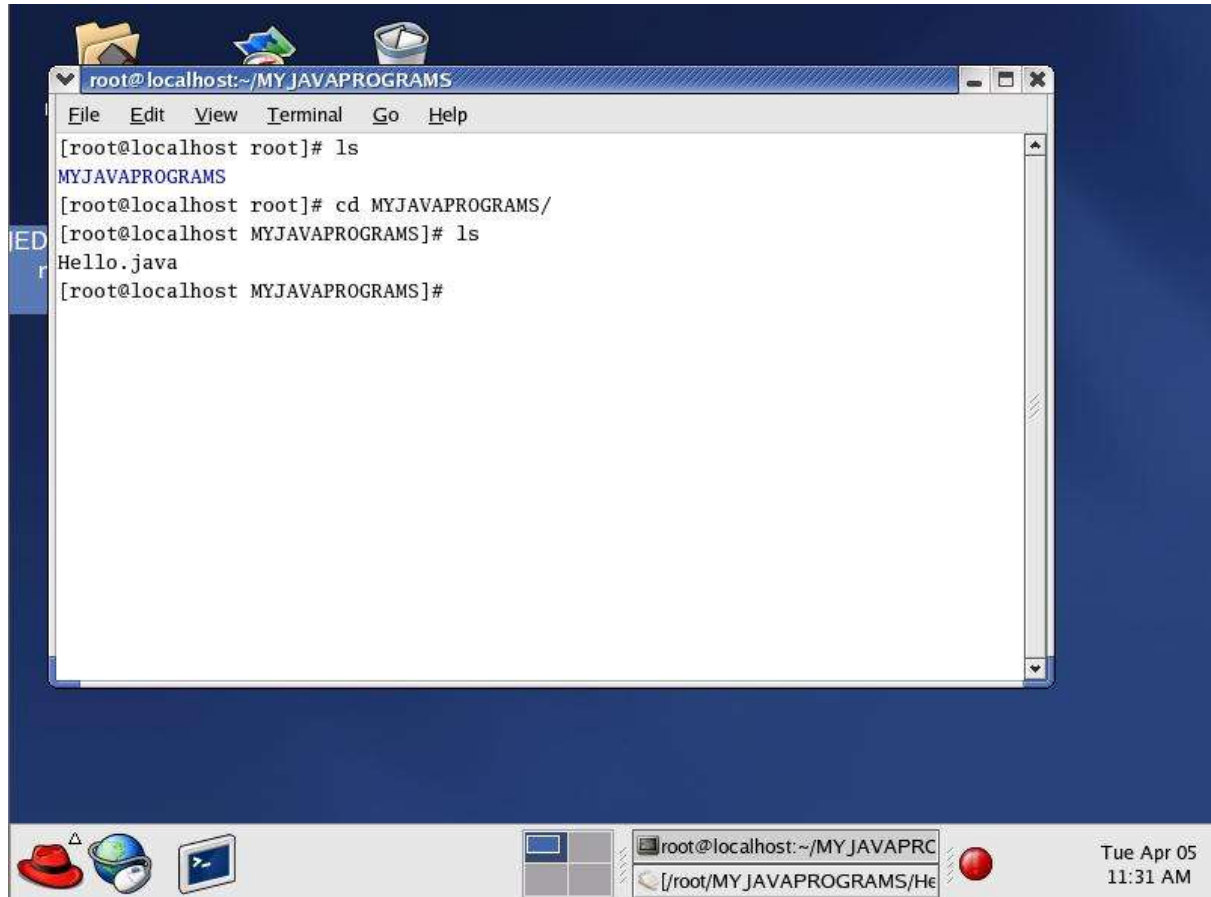


Now, you can see here that there is a folder named "MYJAVAPROGRAMS" which we have created a while ago, and where we saved our Hello.java program. Now let's go inside that directory.

To go inside a directory, you type in the command: **cd [directory name]**. The "cd" command stands for, change directory. In this case, since the name of our directory is MYJAVAPROGRAMS, you type in: **cd MYJAVAPROGRAMS**



Once inside the folder where your Java programs are, let us now start compiling your Java program. Take note that, you should make sure that the file is inside the folder where you are in. In order to do that, execute the "ls" command again to see if your file is inside that folder.

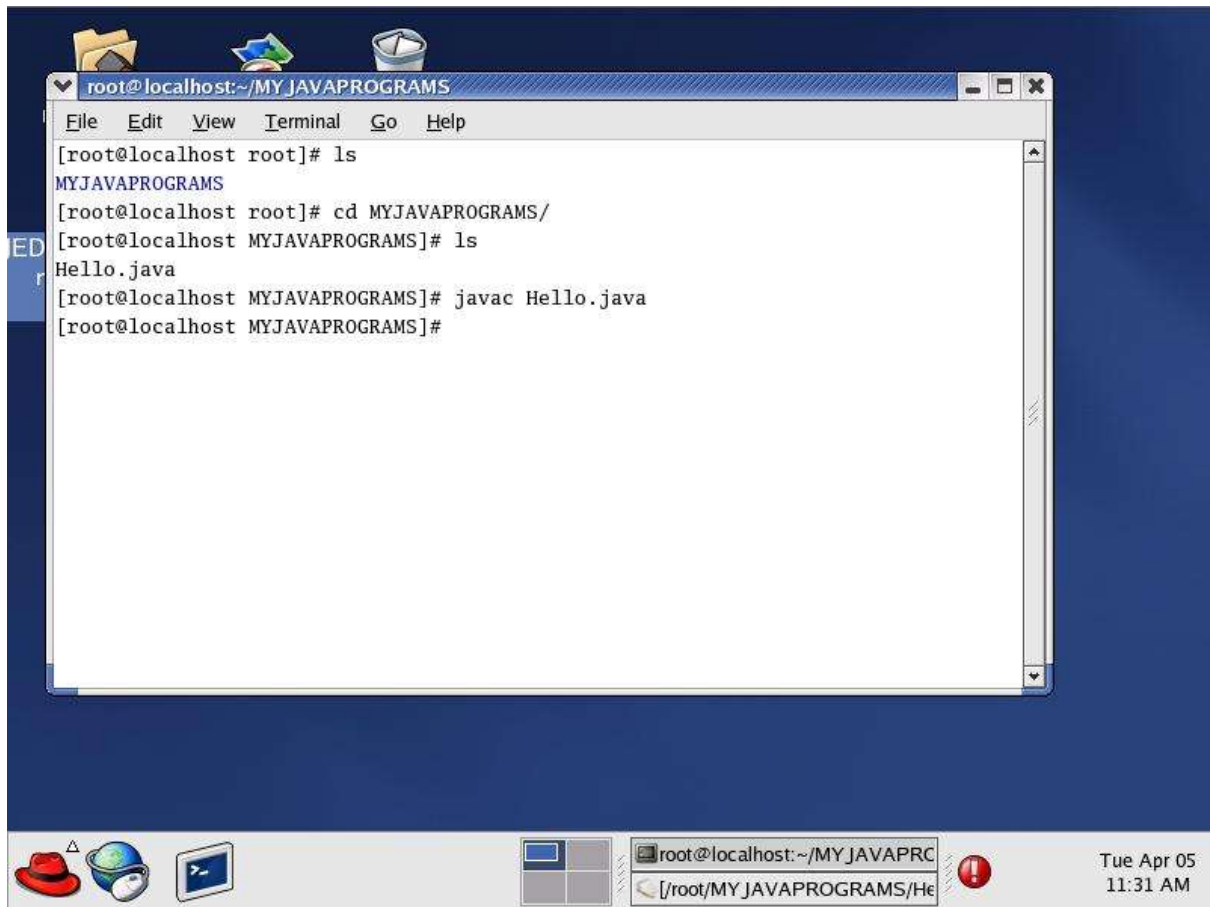


The screenshot shows a terminal window titled "root@localhost:~/MY_JAVAPROGRAMS". The terminal output is as follows:

```
root@localhost root]# ls
MY_JAVAPROGRAMS
root@localhost root]# cd MY_JAVAPROGRAMS/
root@localhost MY_JAVAPROGRAMS]# ls
Hello.java
root@localhost MY_JAVAPROGRAMS]#
```

The terminal window has a menu bar with "File", "Edit", "View", "Terminal", "Go", and "Help". The desktop background is dark blue, and the taskbar at the bottom shows various icons and the system clock indicating "Tue Apr 05 11:31 AM".

To compile a Java program, we type in the command: **javac [filename]**. So in this case, type in: **javac Hello.java**.

A screenshot of a Linux desktop environment with a dark blue background. A terminal window is open, titled 'root@localhost:~/MY_JAVAPROGRAMS'. The terminal shows the following commands and output: [root@localhost root]# ls MY_JAVAPROGRAMS, [root@localhost root]# cd MY_JAVAPROGRAMS/, [root@localhost MY_JAVAPROGRAMS]# ls Hello.java, [root@localhost MY_JAVAPROGRAMS]# javac Hello.java, and [root@localhost MY_JAVAPROGRAMS]#. The terminal window has a menu bar with 'File', 'Edit', 'View', 'Terminal', 'Go', and 'Help'. The desktop has a taskbar at the bottom with icons for a red hat, a globe, a terminal, and a window manager. The system tray shows the user 'root@localhost:~/MY_JAVAPRC', the current directory '/root/MY_JAVAPROGRAMS/He', and the date and time 'Tue Apr 05 11:31 AM'.

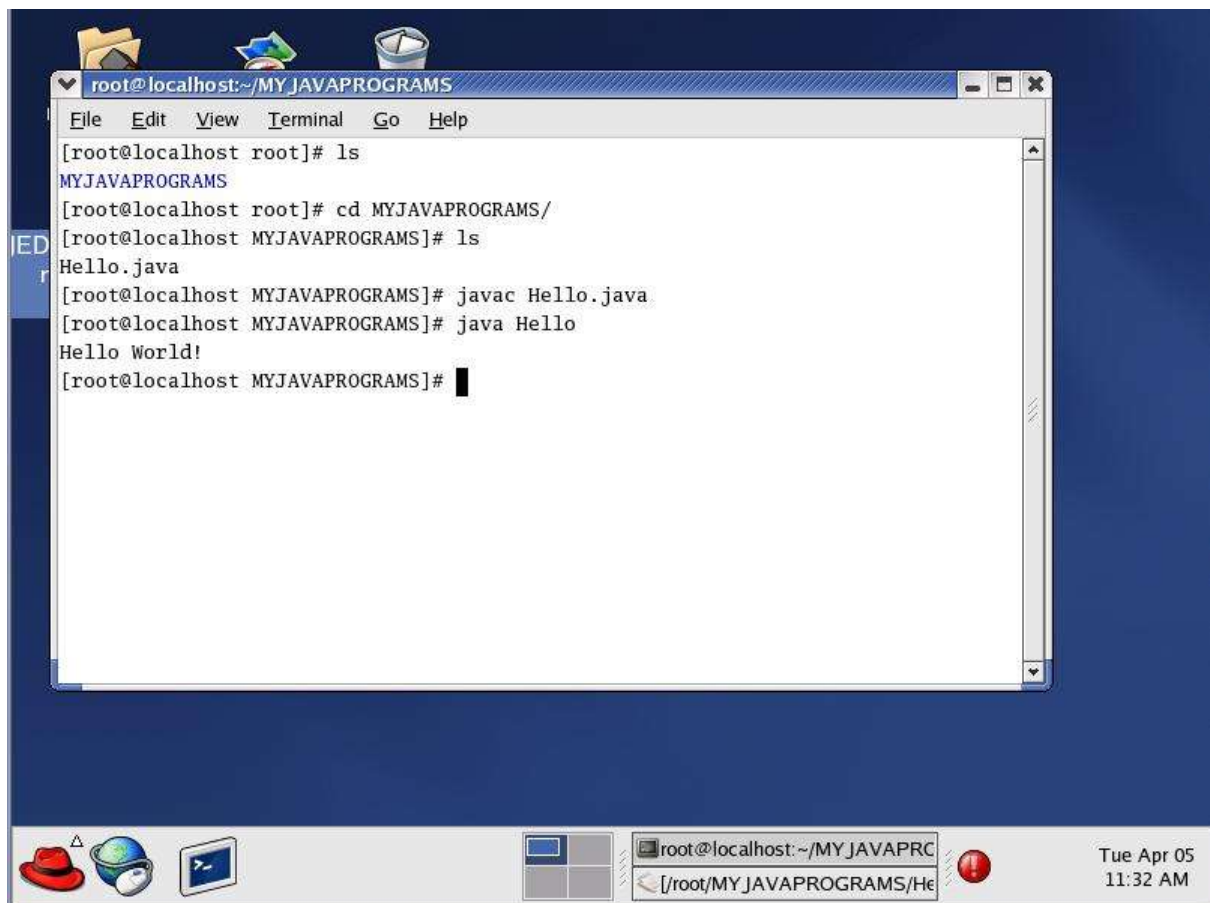
During compilation, javac adds a file to the disk called **[filename].class**, or in this case, **Hello.class**, which is the actual bytecode.

Step 6: Running the Program

Now, assuming that there are no problems during compilation (we'll explore more of the problems encountered during compilation in the next section), we are now ready to run your program.

To run your Java program, type in the command: **java [filename without the extension]**, so in the case of our example, type in: **java Hello**

You can see on the screen that you have just run your first Java program that prints the message, "Hello world!".

A screenshot of a Linux desktop environment with a dark blue background. A terminal window titled 'root@localhost:~/MY_JAVAPROGRAMS' is open, displaying the following commands and output:

```
[root@localhost root]# ls
MY_JAVAPROGRAMS
[root@localhost root]# cd MY_JAVAPROGRAMS/
[root@localhost MY_JAVAPROGRAMS]# ls
Hello.java
[root@localhost MY_JAVAPROGRAMS]# javac Hello.java
[root@localhost MY_JAVAPROGRAMS]# java Hello
Hello World!
[root@localhost MY_JAVAPROGRAMS]#
```

The terminal window has a menu bar with 'File', 'Edit', 'View', 'Terminal', 'Go', and 'Help'. The desktop features several icons: a folder, a trash can, a red hat, a globe, and a terminal icon. The taskbar at the bottom shows the terminal window's title bar, a task switcher, and the system clock indicating 'Tue Apr 05 11:32 AM'.

3.2.1 Errors

What we've shown so far is a Java program wherein we didn't encounter any problems in compiling and running. However, this is not always the case. As what we have discussed in the first part of this course, we usually encounter errors along the way.

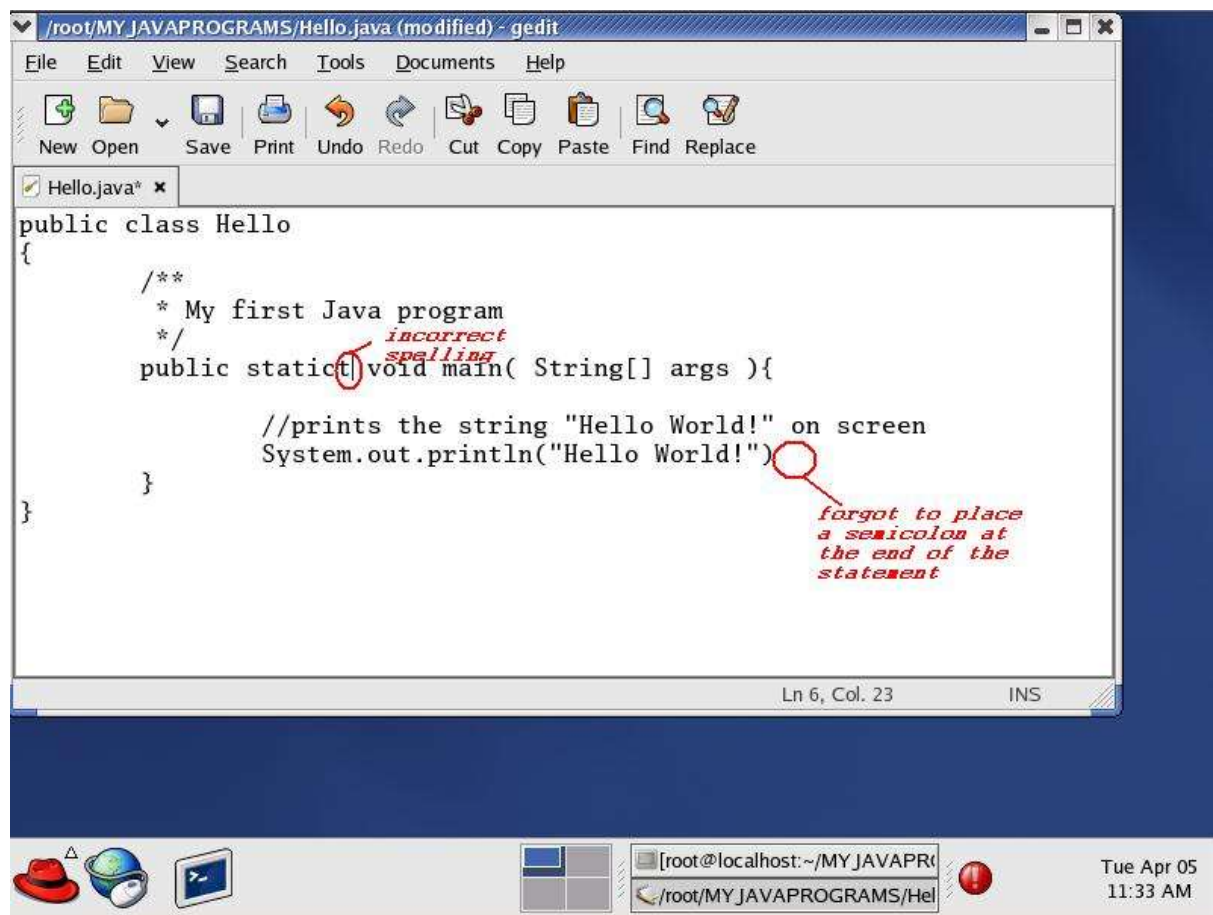
As discussed before, there are two types of errors. The first one is a compile-time error or also called as syntax error. The second one is the runtime error.

3.2.1.1 Syntax Errors

Syntax errors are usually typing errors. You may have misspelled a command in Java or forgot to write a semi-colon at the end of a statement. Java attempts to isolate the error by displaying the line of code and pointing to the first incorrect character in that line. However, the problem may not be at the exact point.

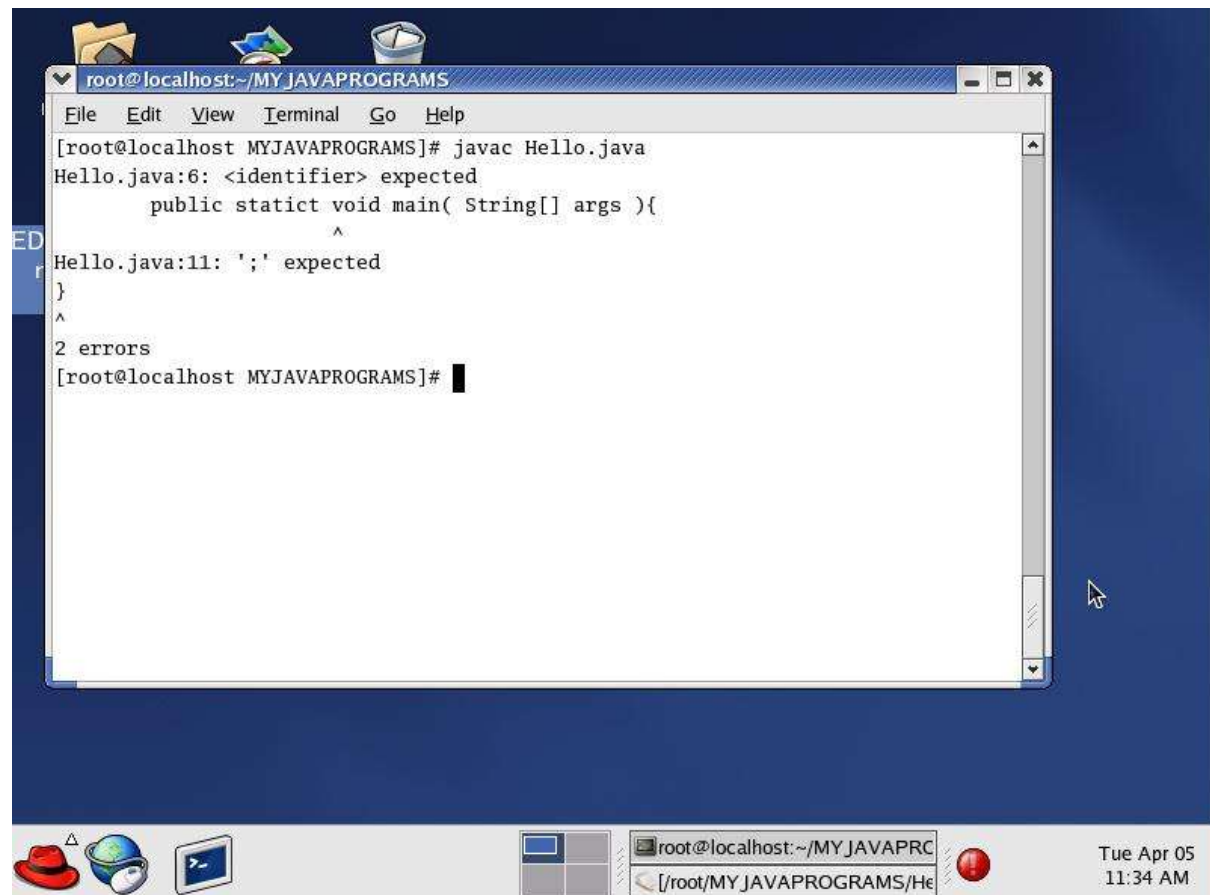
Other common mistakes are in capitalization, spelling, the use of incorrect special characters, and omission of correct punctuation.

Let's take for example, our Hello.java program wherein we intentionally omit the semicolon at one statement and we try to type the incorrect spelling of a command.



See the error messages generated after compiling the program. The first error message suggests that there is an error in line 6 of your program. It pointed to the next word after the **statict**, which should be spelled as static.

The second error message suggests that there is a missing semicolon after your statement.

A screenshot of a terminal window titled 'root@localhost:~/MY_JAVAPROGRAMS'. The window shows the output of the 'javac Hello.java' command. The first error message is 'Hello.java:6: <identifier> expected' with an arrow pointing to the word 'statict' in the line 'public statict void main(String[] args){'. The second error message is 'Hello.java:11: ';' expected' with an arrow pointing to the closing brace '}' in the line '}'. Below the error messages, it says '2 errors' and then '[root@localhost MYJAVAPROGRAMS]#'. The terminal window has a menu bar with 'File', 'Edit', 'View', 'Terminal', 'Go', and 'Help'. The desktop background is dark blue with icons for a folder, a trash can, and a terminal. The taskbar at the bottom shows a red hat icon, a globe icon, a terminal icon, and a status bar with the text 'root@localhost:~/MY_JAVAPROGRAMS' and 'Tue Apr 05 11:34 AM'.

As a rule of thumb, if you encounter a lot of error messages, try to correct the first mistake in a long list, and try to compile the program again. Doing so may reduce the total number of errors dramatically.

3.2.1.2 Run-time Errors

Run-time errors are errors that will not display until you run or execute your program. Even programs that compile successfully may display wrong answers if the programmer has not thought through the logical processes and structures of the program.

3.3 Using Netbeans

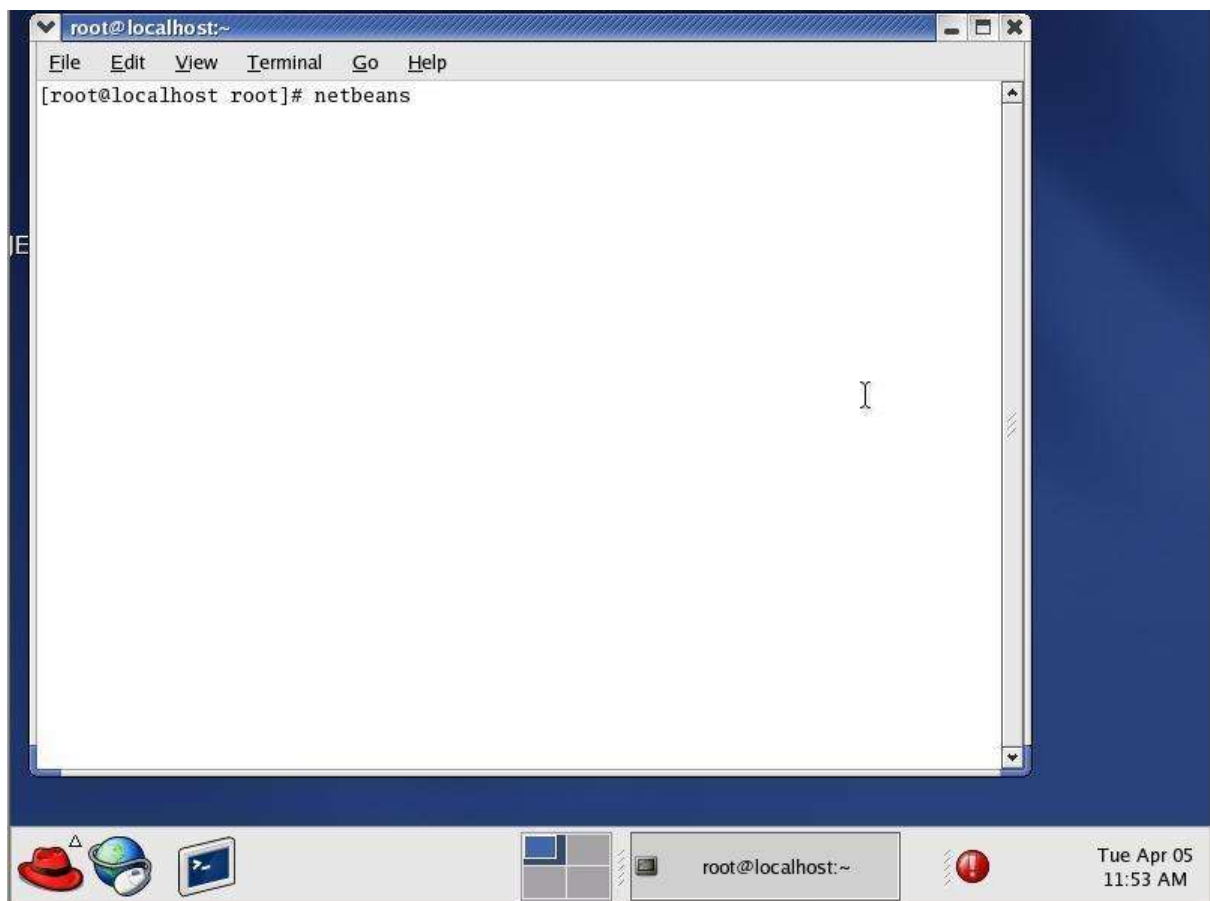
Now that we've tried doing our programs the complicated way, let's now see how to do all the processes we've described in the previous sections by using just one application.

In this part of the lesson, we will be using **Netbeans**, which is an **Integrated Development Environment or IDE**. An IDE is a programming environment integrated into a software application that provides a GUI builder, a text or code editor, a compiler and/or interpreter and a debugger.

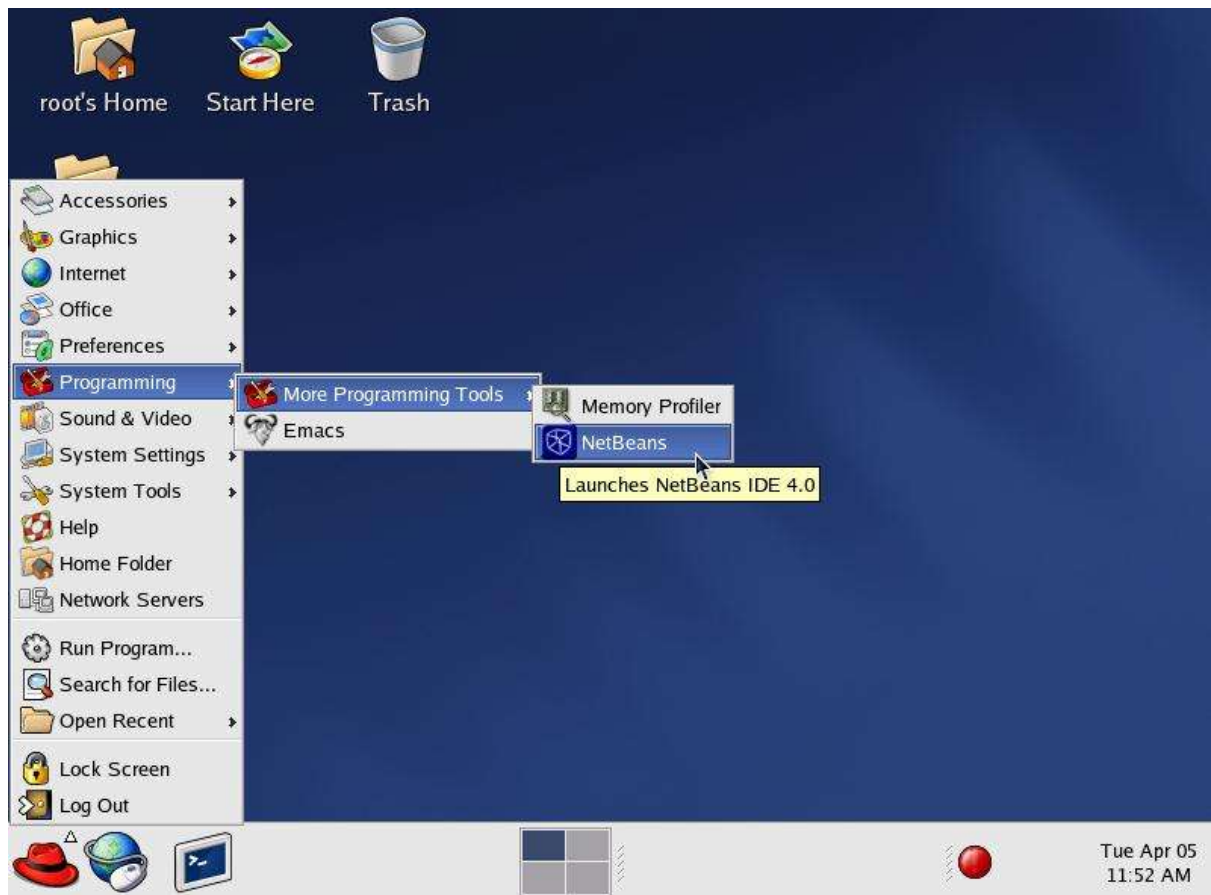
Step 1: Run Netbeans

There are two ways to run Netbeans. One is through command-line using terminal, or by jst clicking on the shortcut button found on the main menu.

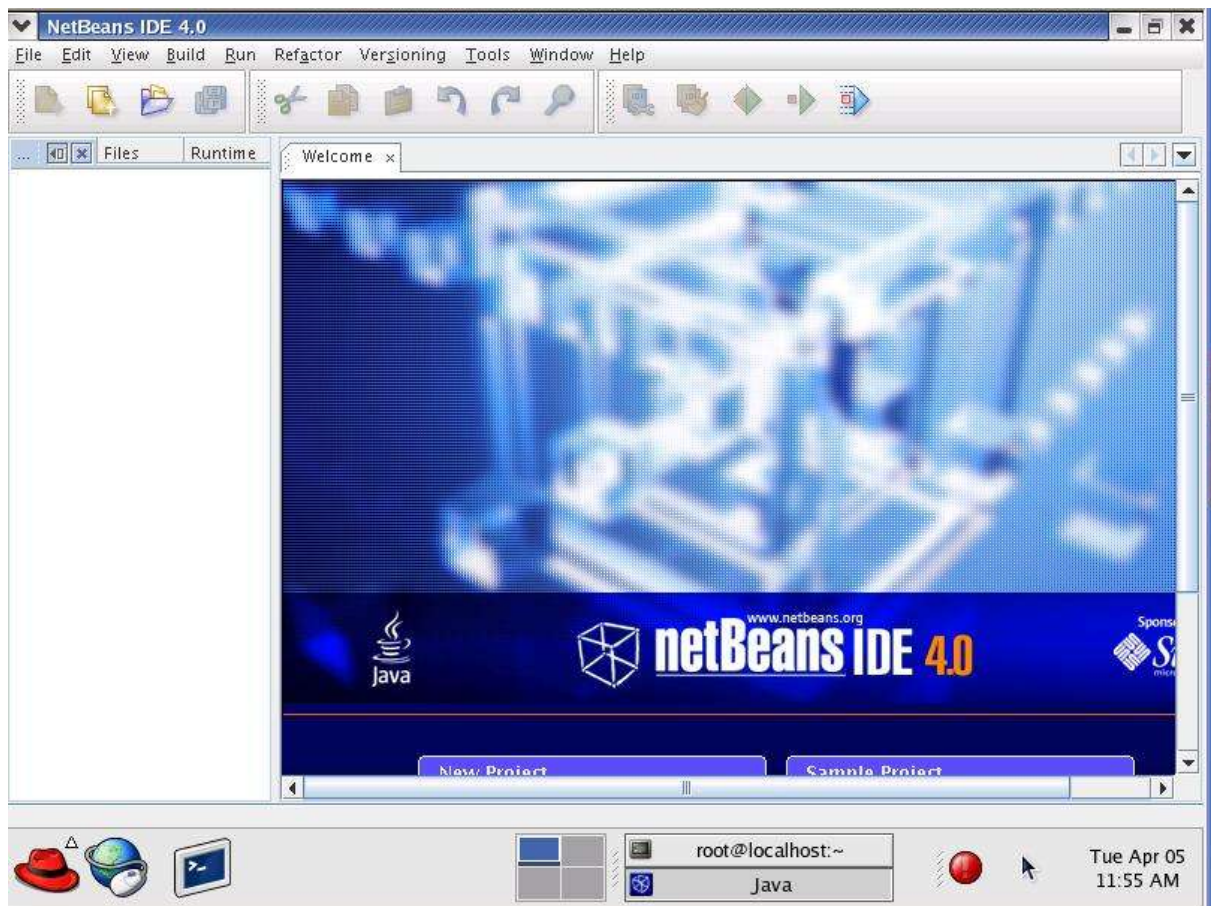
To run Netbeans using command-line. Open terminal (see steps on how to run terminal in the previous discussion), and type: **netbeans**



The second way to run Netbeans, is by clicking on Menu-> Programming-> More Programming Tools-> Netbeans.

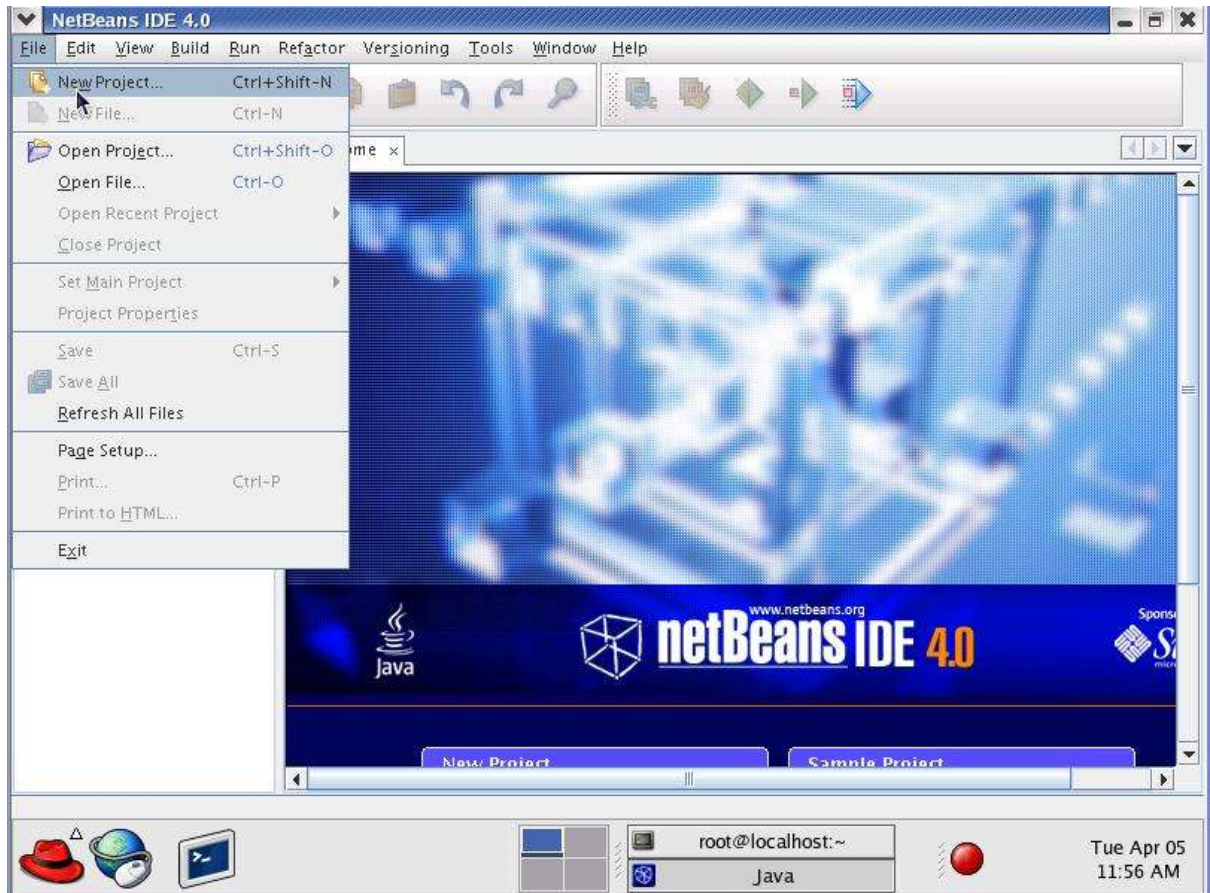


After you've open NetBeans IDE, you will see a graphical user interface (GUI) similar to what is shown below.

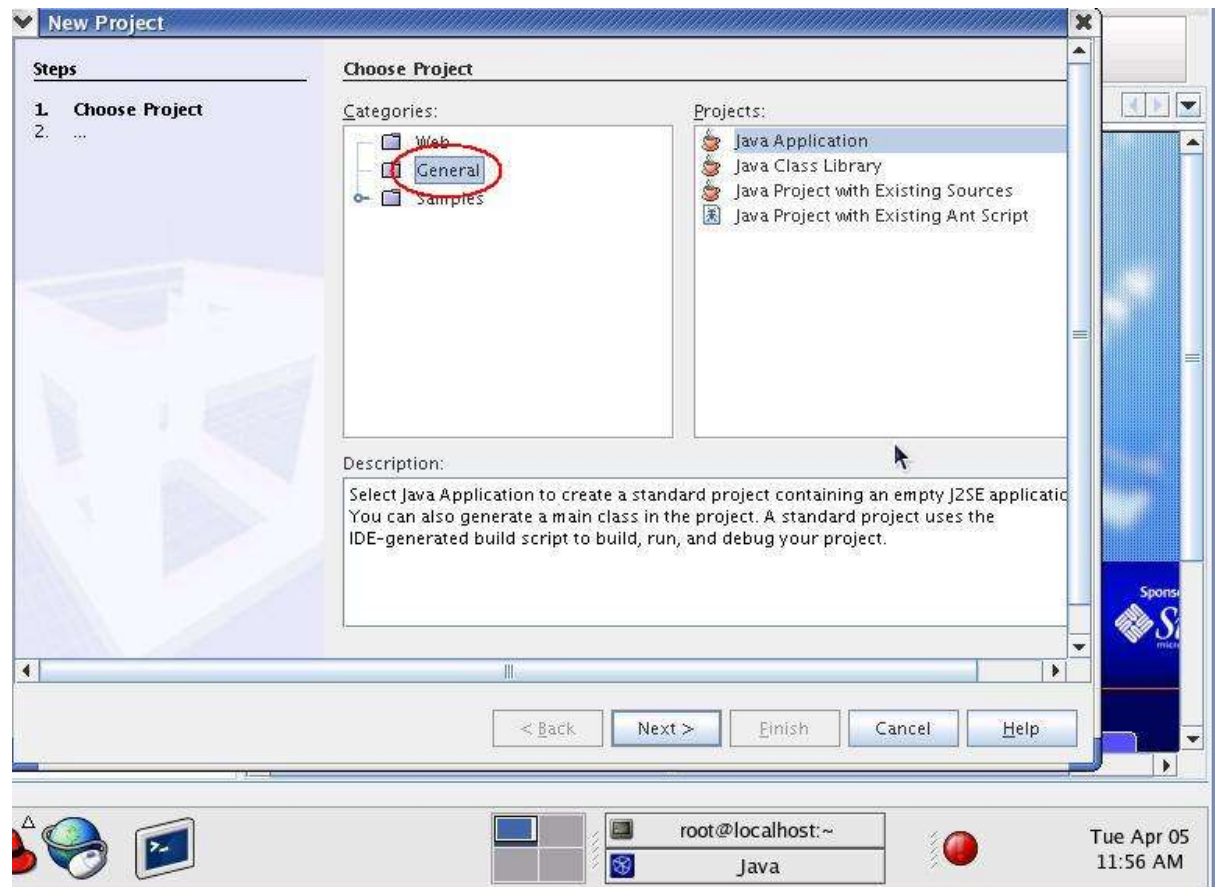


Step 2: Make a project

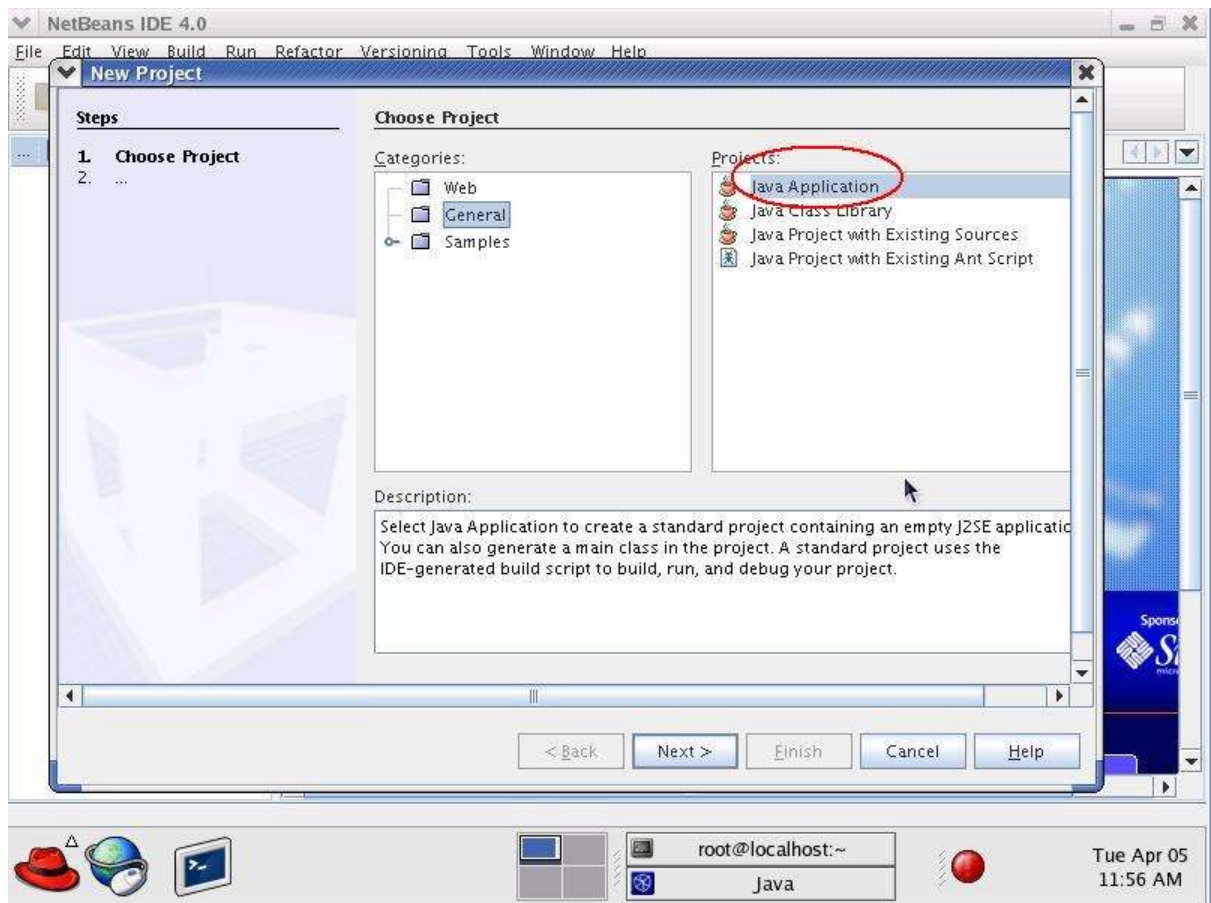
Now, let's first make a project. Click on File-> New Project.



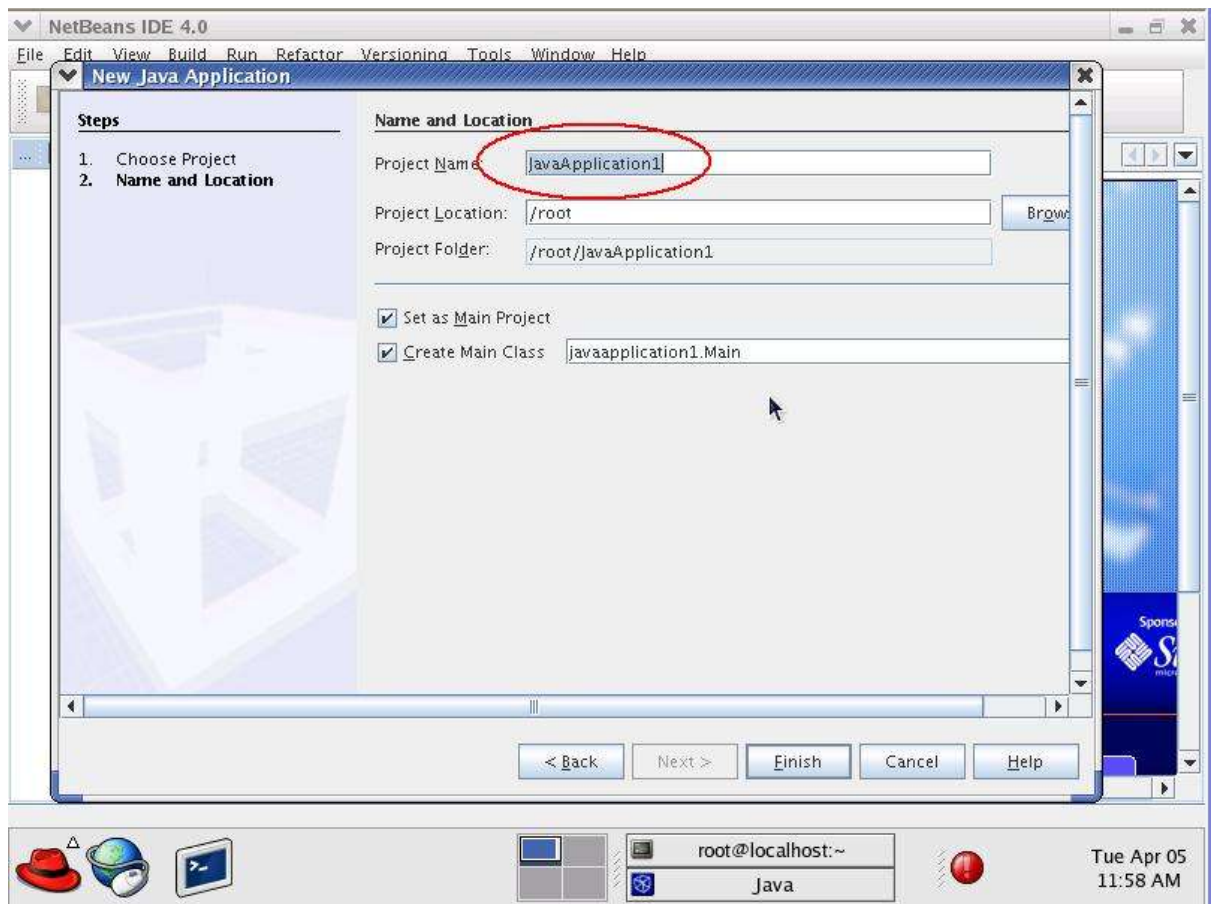
After doing this, a New Project dialog will appear.



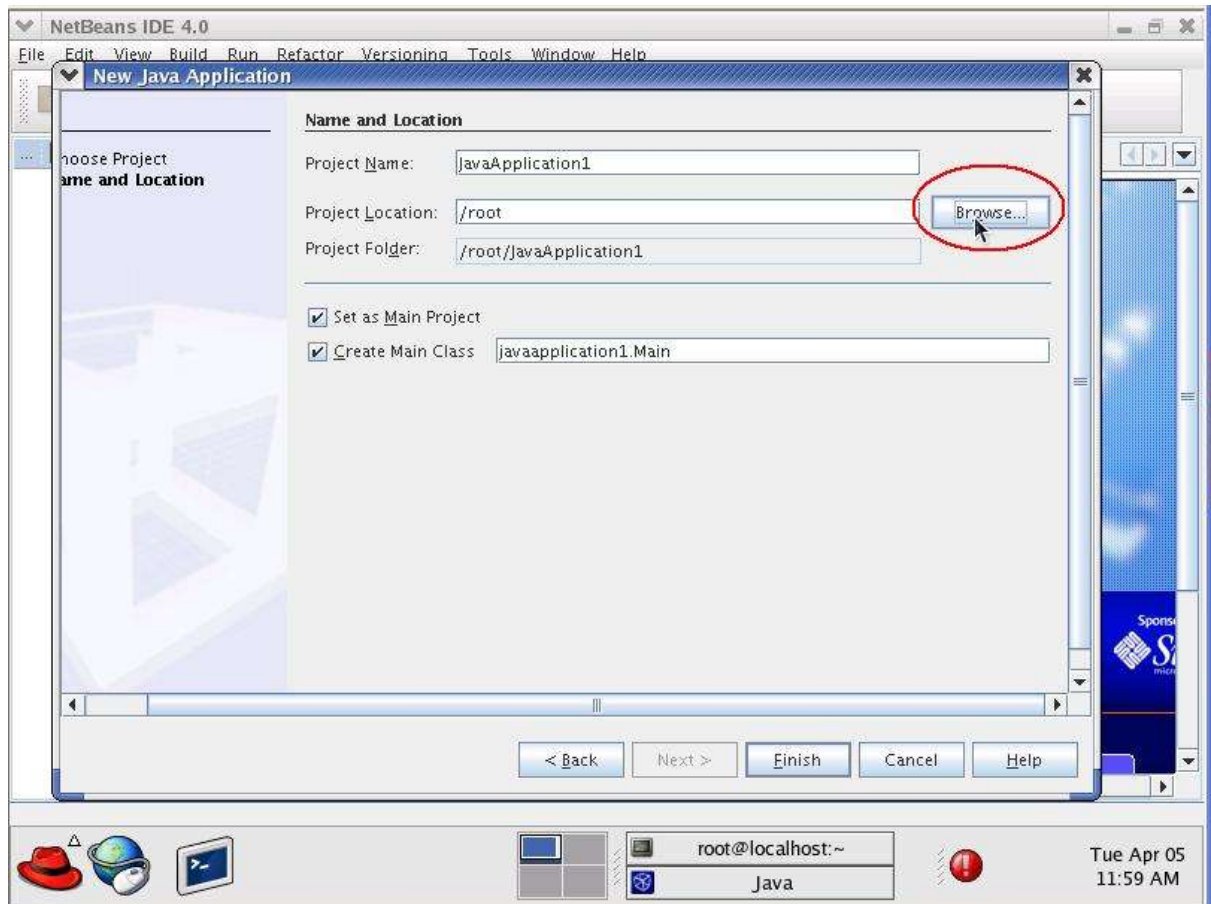
Now click on Java Application and click on the NEXT button.



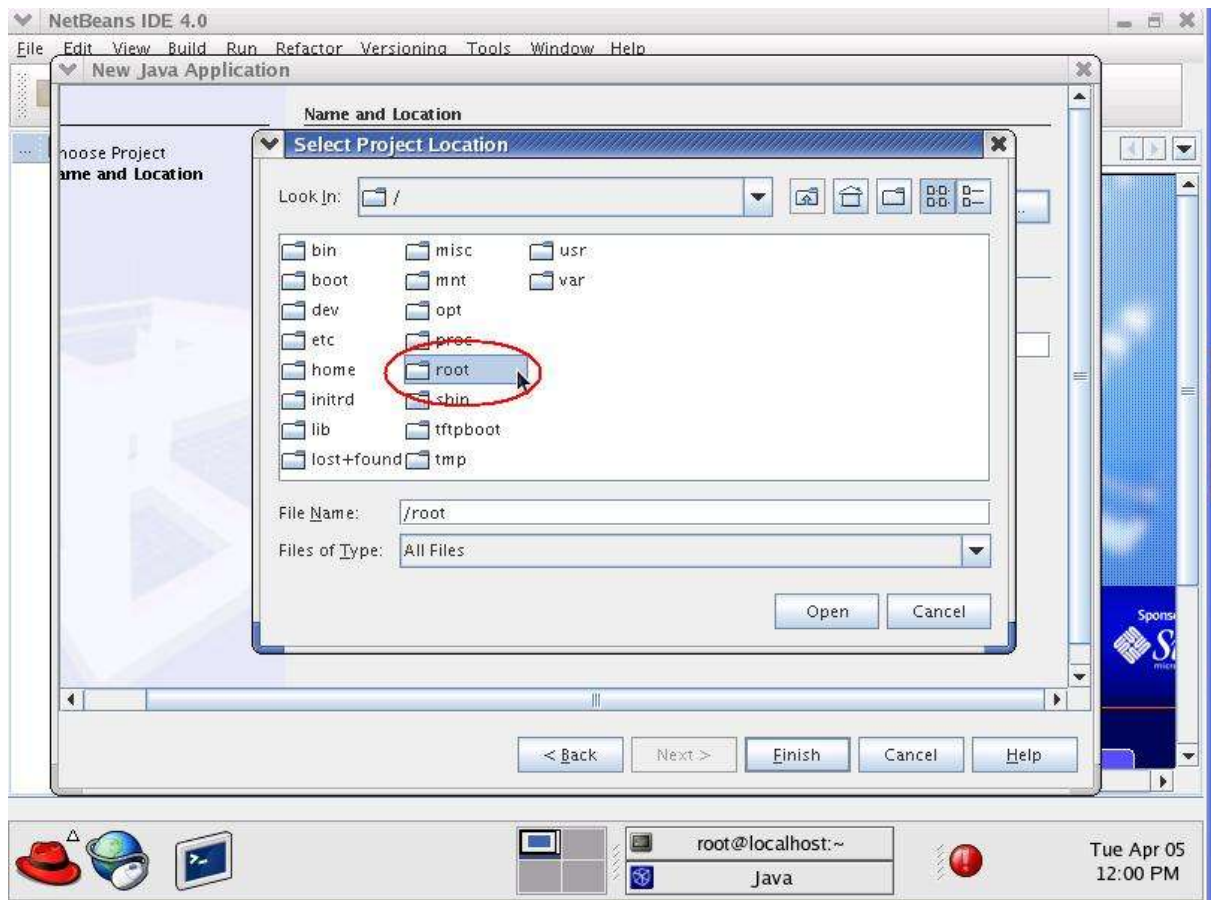
Now, a New Application dialog will appear. Edit the Project Name part and type in "HelloApplication".



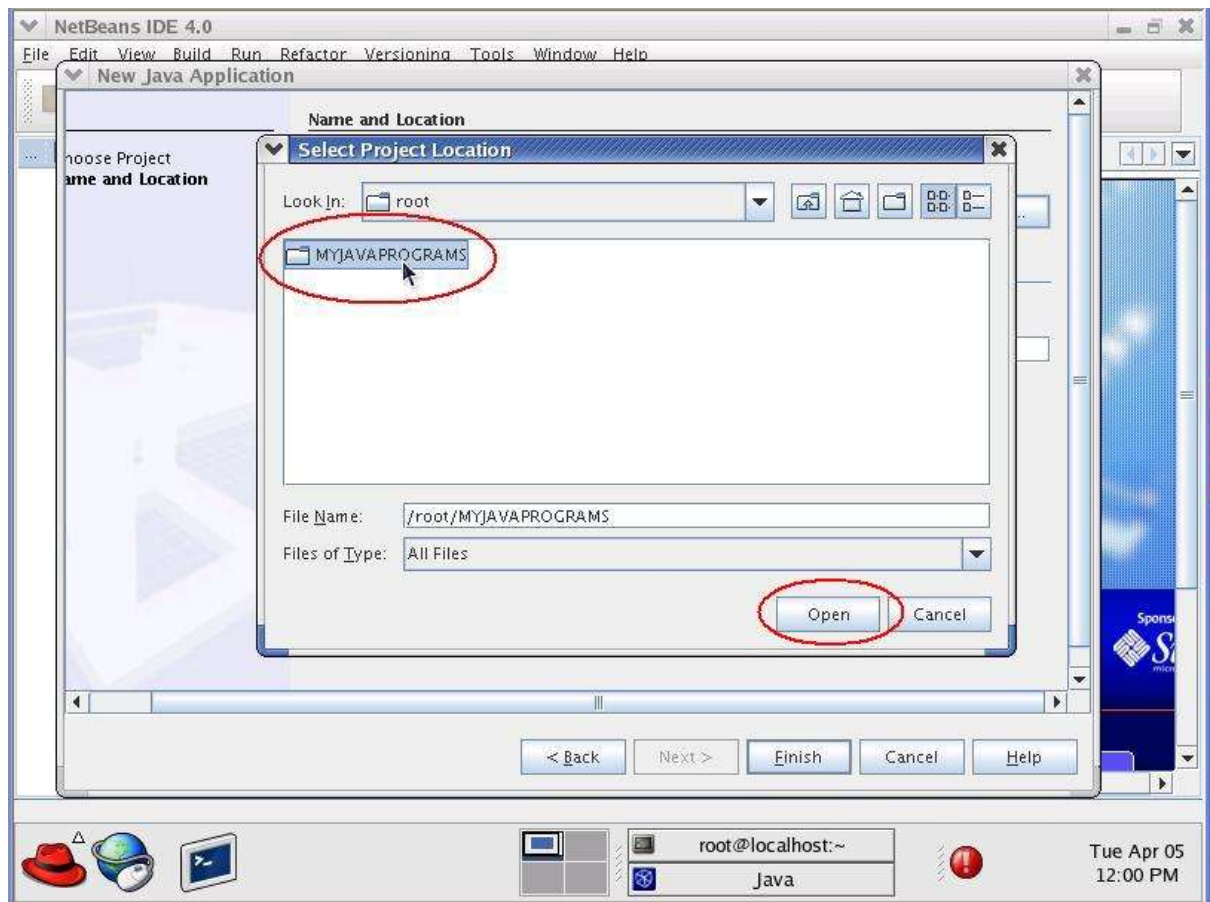
Now try to change the Application Location, by clicking on the BROWSE button.



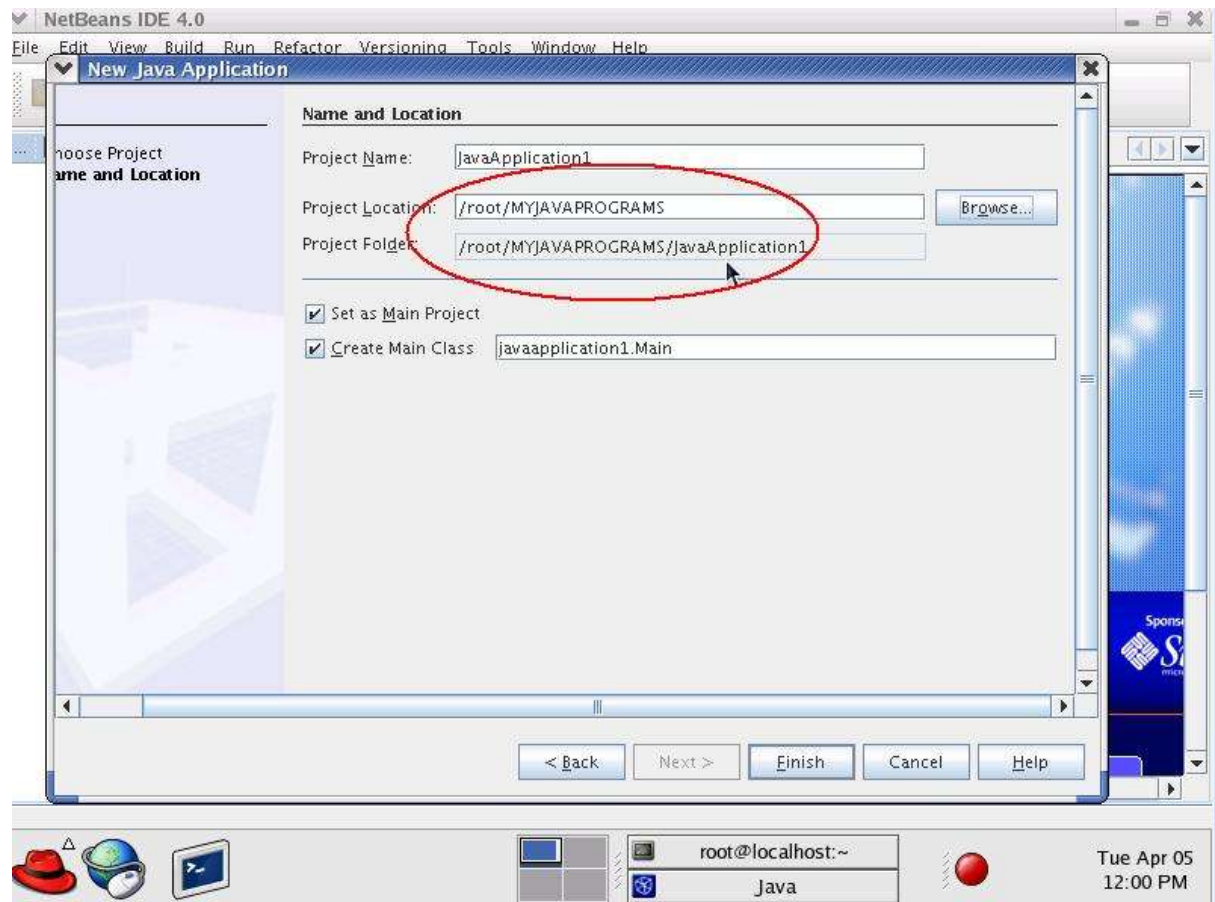
A Select Project Location dialog will then appear. Double-click on the root folder.



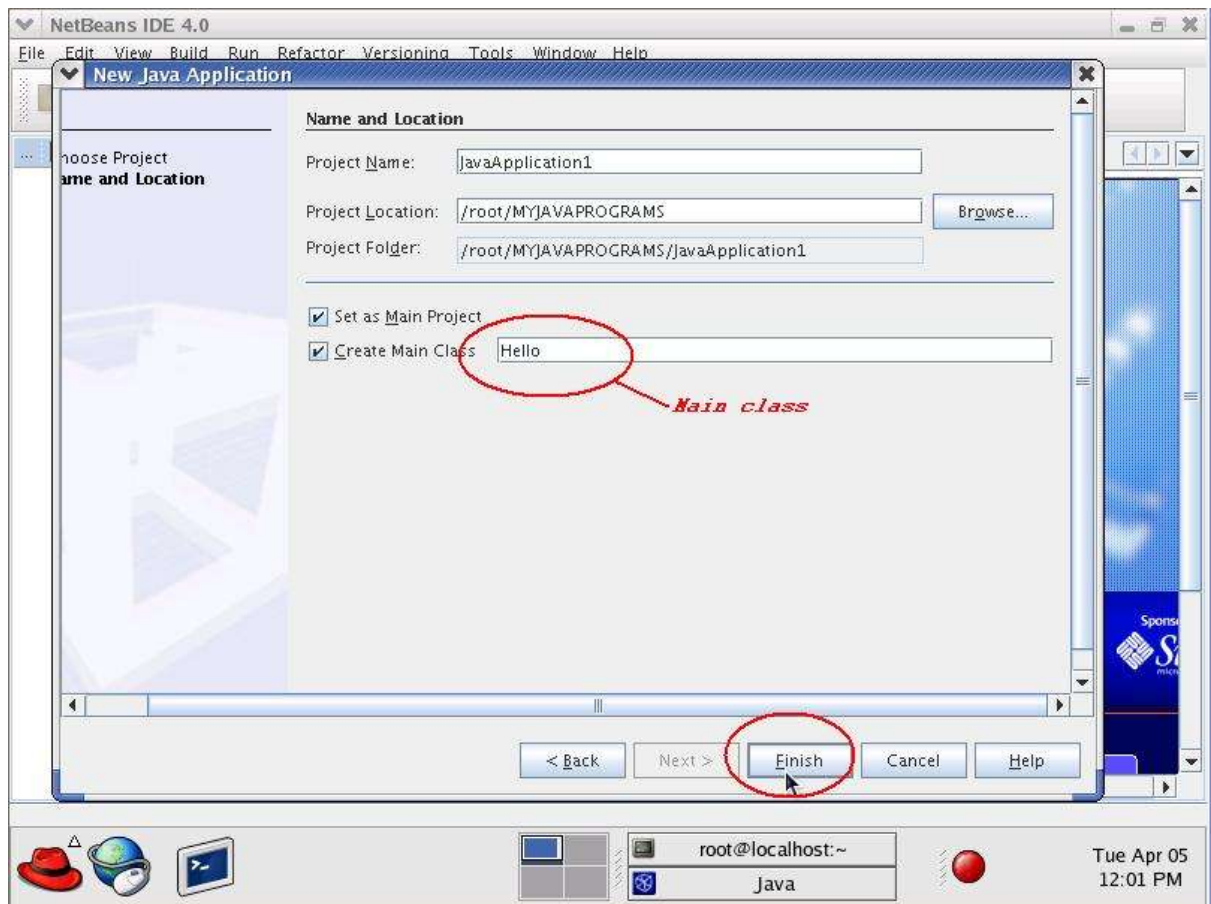
The contents of the root folder is then displayed. Now double-click on the MYJAVAPROGRAMS folder and click on the OPEN button.



See now that the Project Location and Project Folder is changed to /root/MYJAVAPROGRAMS.



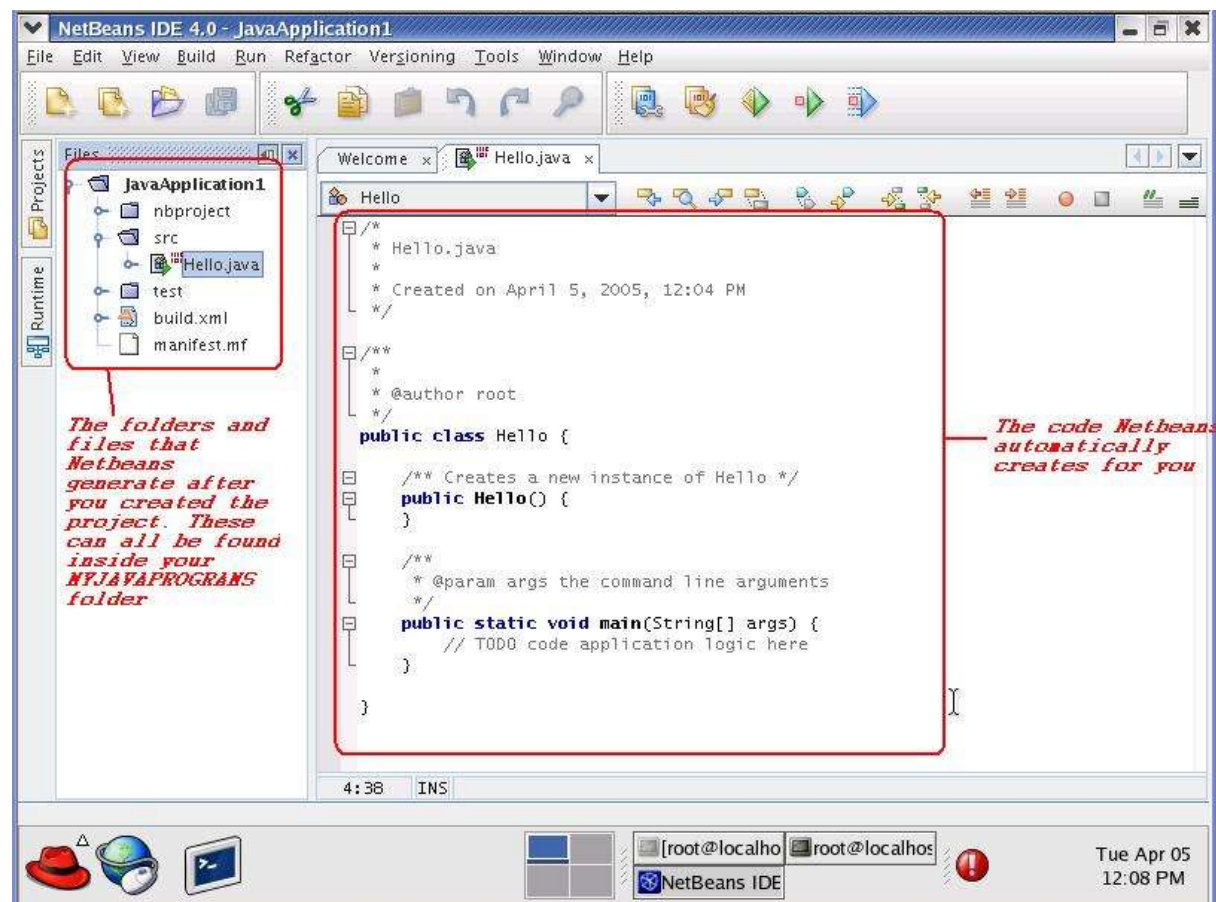
Finally, on the Create Main Class textfield, type in Hello as the main class' name, and then click on the FINISH button.



Step 3: Type in your program

Before typing in your program, let us first describe the main window after creating the project.

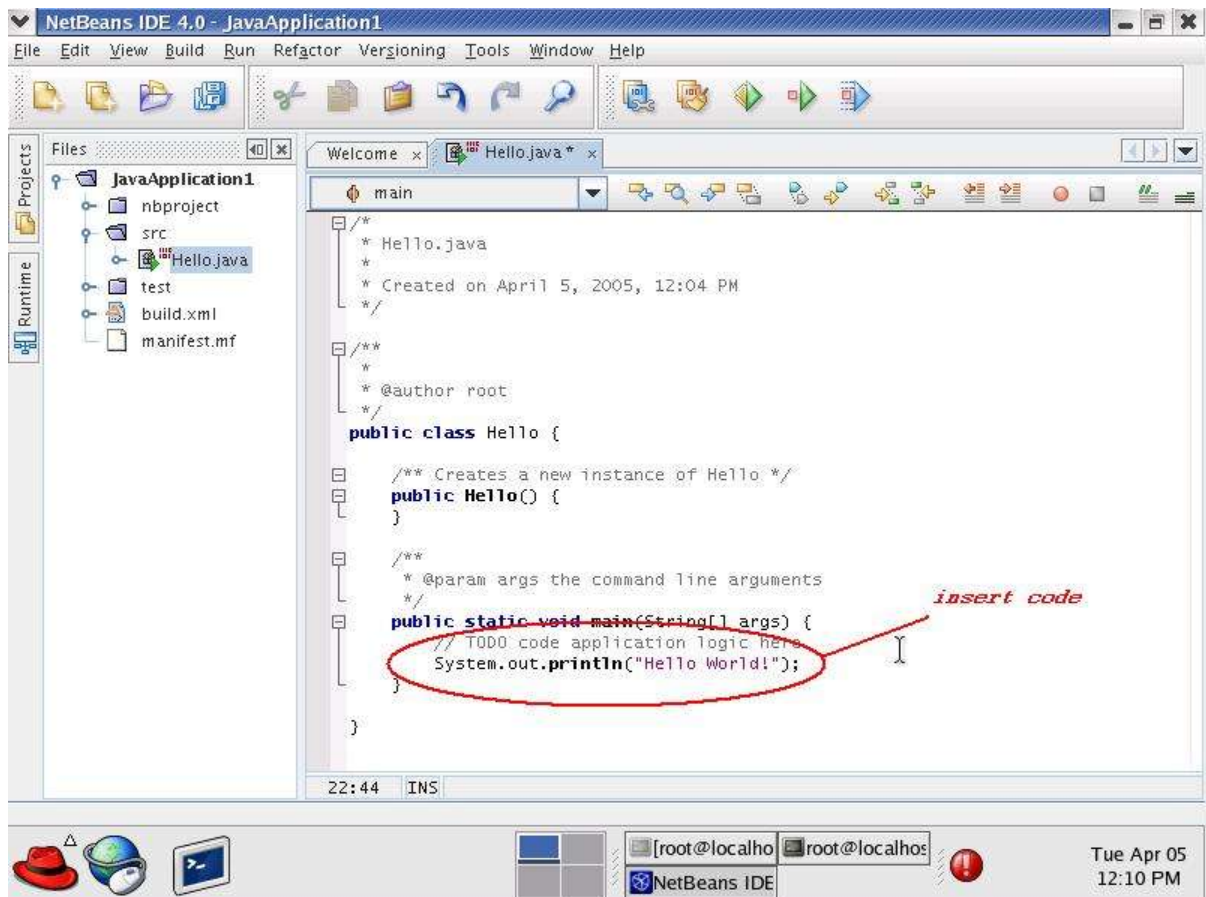
As shown below, NetBeans automatically creates the basic code for your Java program. You can just add your own statements to the generated code. On the left side of the window, you can see a list of folders and files that NetBeans generated after creating the project. This can all be found in your MYJAVAPROGRAMS folder, where you set the Project location.



Now, try to modify the code generated by Netbeans. Ignore the other parts of the program for now, as we will explain the details of the code later. Insert the code:

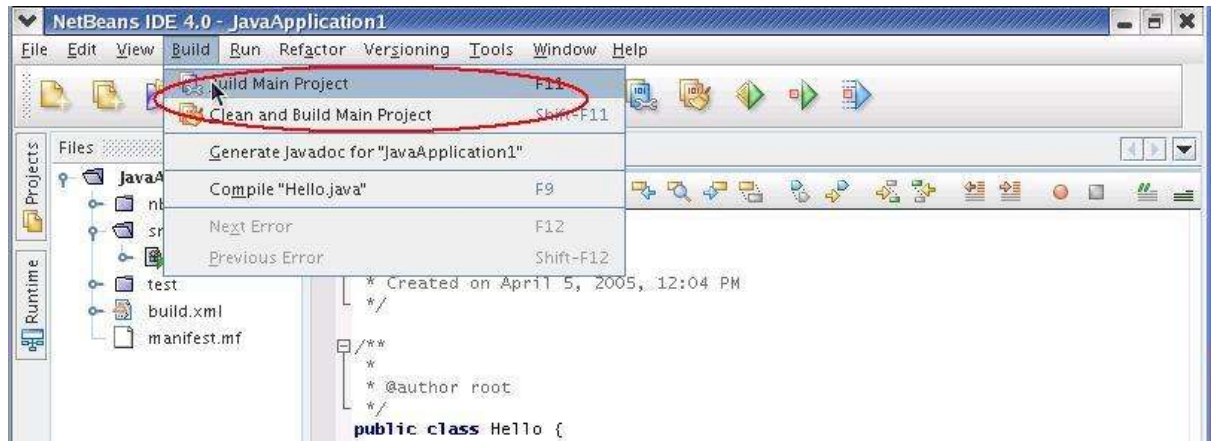
```
System.out.println("Hello world!");
```

after the statement, `//TODO code application logic here.`



Step 4: Compile your program

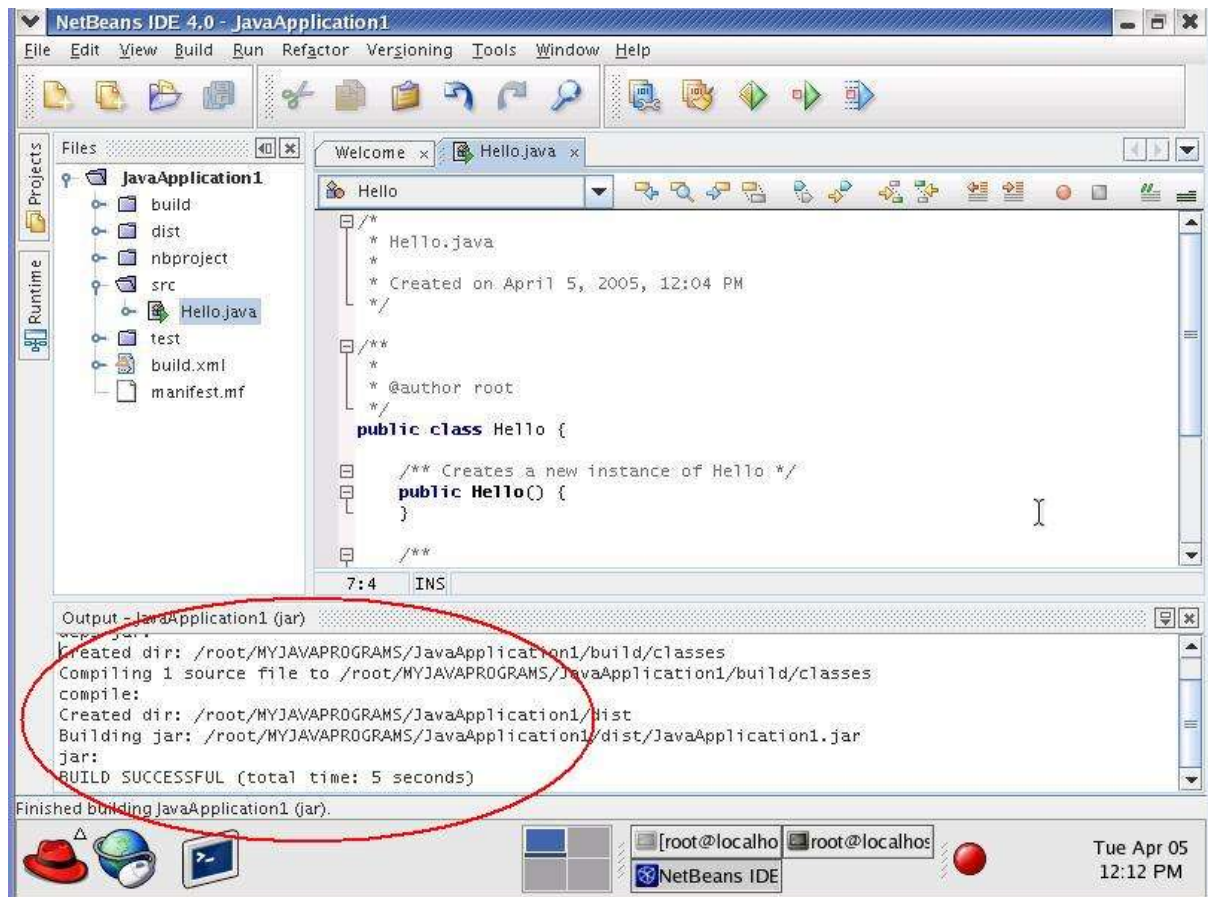
Now, to compile your program, just click on Build -> Build Main Project.



Or, you could also use the shortcut button to compile your code.

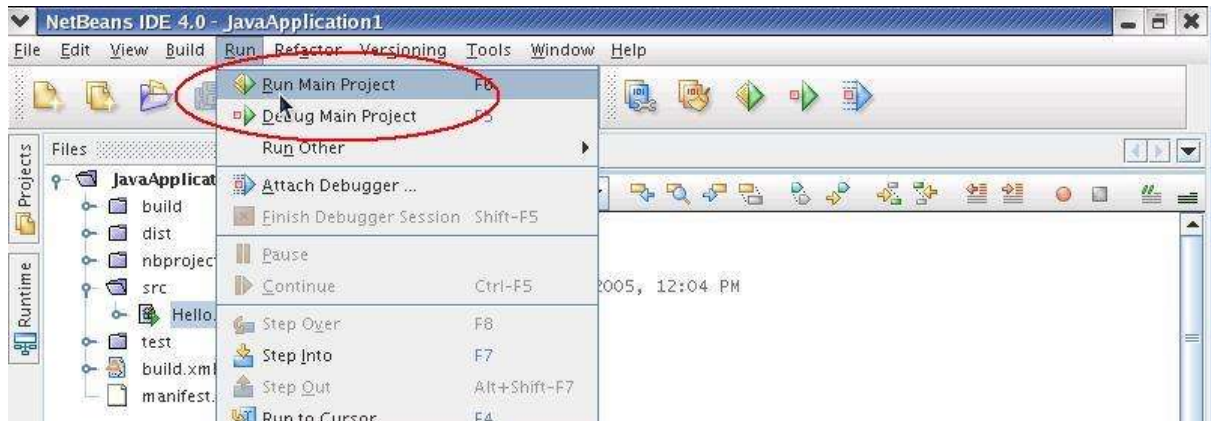


If there are no errors in your program, you will see a build successful message on the output window.

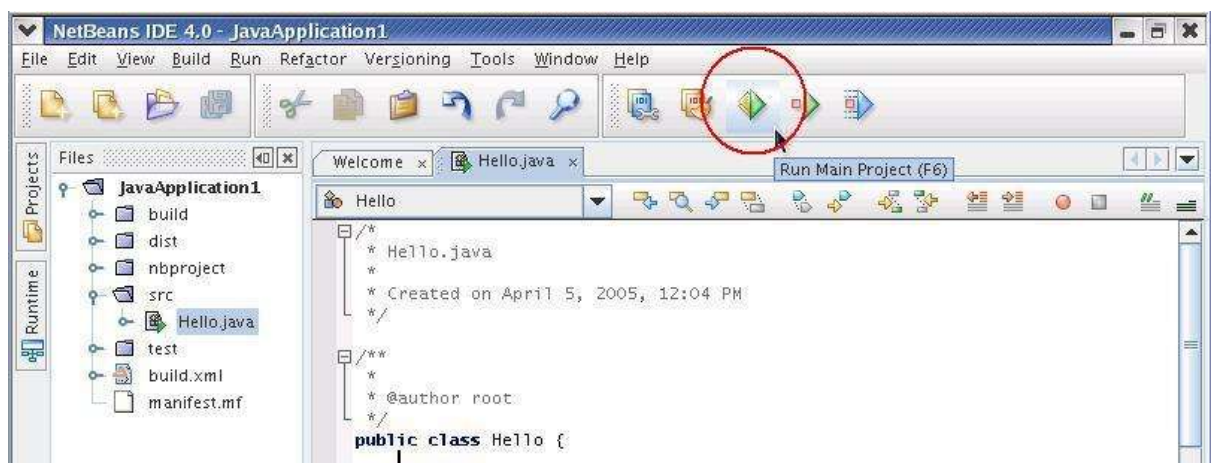


Step 5: Run your program

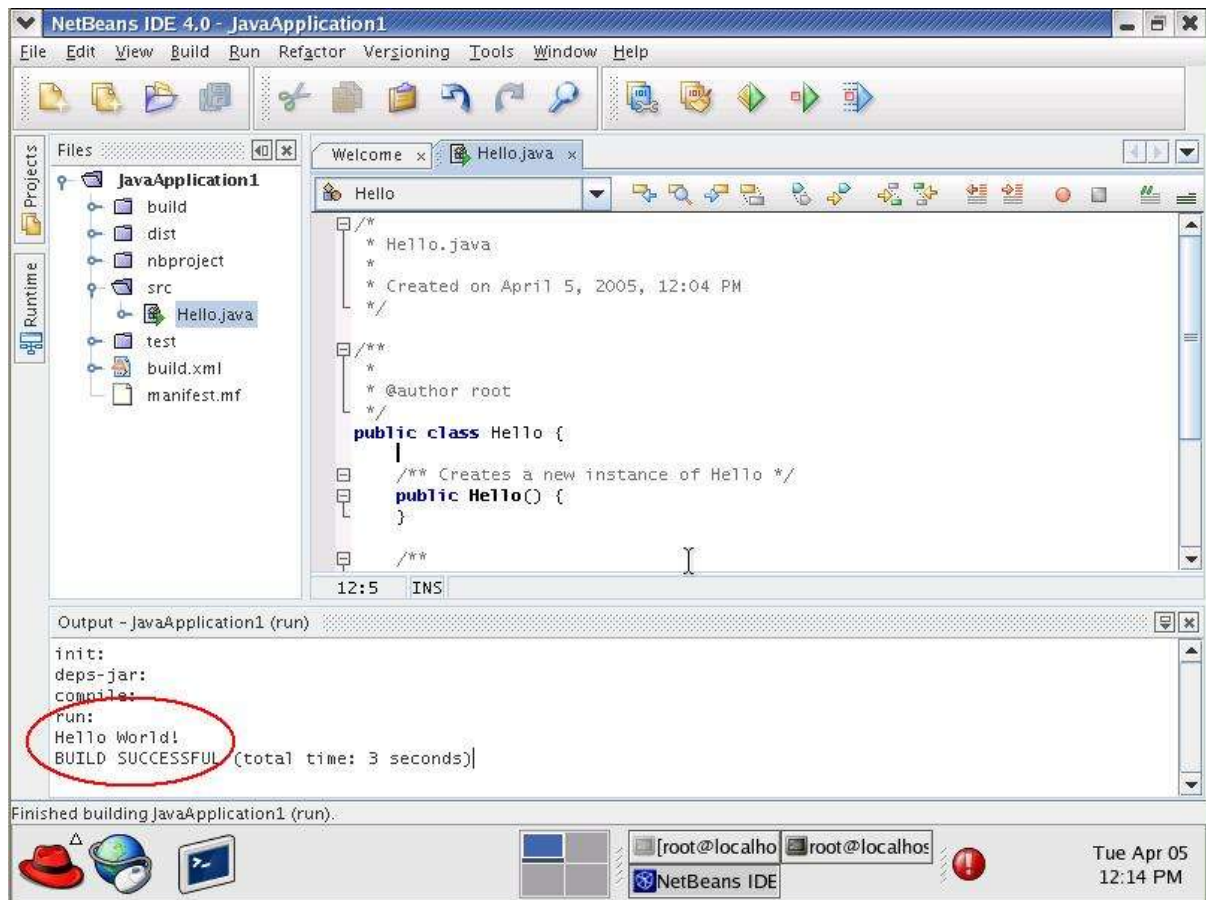
To run your program, click on Run-> Run Main Project.



Or you could also use the shortcut button to run your program.



The output of your program is displayed in the output window.



3.4 Exercises

3.4.1 Hello World!

Using Netbeans, create a class named: [YourName]. The program should output on the screen:

```
Welcome to Java Programming [YourName]!!!
```

3.4.2 The Tree

Using Netbeans, create a class named: **TheTree**. The program should output the following lines on the screen:

```
I think that I shall never see,  
a poem as lovely as a tree.  
A tree whose hungry mouth is pressed  
Against the Earth's sweet flowing breast.
```

4 Programming Fundamentals

In this section, we will be discussing the basic parts of a Java program. We will start by trying to explain the basic parts of the Hello.java program introduced in the previous section. We will also be discussing some coding guidelines or code conventions along the way to help in effectively writing readable programs.

4.1 *Dissecting my first Java program*

Now, we'll try to dissect your first Java program:

```
public class Hello
{
    /**
     * My first java program
     */
    public static void main(String[] args) {

        //prints the string "Hello world" on screen
        System.out.println("Hello world!");
    }
}
```

The first line of the code,

```
public class Hello
```

indicates the name of the class which is **Hello**. In Java, all code should be placed inside a class declaration. We do this by using the **class** keyword. In addition, the class uses an *access specifier* **public**, which indicates that our class is accessible to other classes from other packages (packages are a collection of classes). We will be covering packages and access specifiers later.

The next line which contains a curly brace { indicates the start of a block. In this code, we placed the curly brace at the next line after the class declaration, however, we can also place this next to the first line of our code. So, we could actually write our code as:

```
public class Hello
{
or
public class Hello {
```


The next three lines indicates a Java comment. A comment is something used to document a part of a code. It is not part of the program itself, but used for documentation purposes. It is good programming practice to add comments to your code.

```
/**
 * My first java program
 */
```

A comment is indicated by the delimiters `/*` and `*/`. Anything within these delimiters are ignored by the Java compiler, and are treated as comments.

The next line,

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

or can also be written as,

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

indicates the name of one method in **Hello** which is the **main** method. The **main** method is the starting point of a Java program. All programs except Applets written in Java start with the **main** method. Make sure to follow the exact signature.

The next line is also a Java comment,

```
//prints the string "Hello world" on screen
```

Now, we learned two ways of creating comments. The first one is by placing the comment inside `/*` and `*/`, and the other one is by writing `//` at the start of the comment.

The next line,

```
System.out.println("Hello world!");
```

prints the text "Hello World!" on screen. The command `System.out.println()`, prints the text enclosed by quotation on the screen.

The last two lines which contains the two curly braces is used to close the **main** method and **class** respectively.

Coding Guidelines:

1. Your Java programs should always end with the **.java** extension.
2. Filenames should **match** the name of your public class. So for example, if the name of your public class is **Hello**, you should save it in a file called **Hello.java**.
3. You should write comments in your code explaining what a certain class does, or what a certain method do.

4.2 Java Comments

Comments are notes written to a code for documentation purposes. Those text are not part of the program and does not affect the flow of the program.

Java supports three types of comments: C++-style single line comments, C-style multiline comments and special javadoc comments.

4.2.1 C++-Style Comments

C++ Style comments starts with `//`. All the text after `//` are treated as comments. For example,

```
// This is a C++ style or single line comments
```

4.2.2 C-Style Comments

C-style comments or also called multiline comments starts with a `/*` and ends with a `*/`. All text in between the two delimiters are treated as comments. Unlike C++ style comments, it can span multiple lines. For example,

```
/* this is an exmaple of a  
   C style or multiline comments */
```

4.2.3 Special Javadoc Comments

Special Javadoc comments are used for generating an HTML documentation for your Java programs. You can create javadoc comments by starting the line with `/**` and ending it with `*/`. Like C-style comments, it can also span lines. It can also contain certain tags to add more information to your comments. For example,

```
/**  
  This is an example of special java doc comments used for \n  
  generating an html documentation. It uses tags like:  
  @author Florence Balagtas  
  @version 1.2  
  */
```

4.3 Java Statements and blocks

A **statement** is one or more lines of code terminated by a semicolon. An example of a single statement is,

```
System.out.println("Hello world");
```

A **block** is one or more statements bounded by an opening and closing curly braces that groups the statements as one unit. Block statements can be nested indefinitely. Any amount of white space is allowed. An example of a block is,

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

Coding Guidelines:

1. In creating blocks, you can place the opening curly brace in line with the statement, like for example,

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

or you can place the curly brace on the next line, like,

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

2. You should indent the next statements after the start of a block, for example,

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

4.4 Java Identifiers

Identifiers are tokens that represent names of variables, methods, classes, etc. Examples of identifiers are: Hello, main, System, out.

Java identifiers are case-sensitive. This means that the identifier: **Hello** is not the same as **hello**. Identifiers must begin with either a letter, an underscore "_", or a dollar sign "\$". Letters may be lower or upper case. Subsequent characters may use numbers 0 to 9.

Identifiers cannot use Java keywords like class, public, void, etc. We will discuss more about Java keywords later.

Coding Guidelines:

1. For names of classes, capitalize the first letter of the class name. For names of methods and variables, the first letter of the word should start with a small letter. For example:

ThisIsAnExampleOfClassName
thisIsAnExampleOfMethodName

2. In case of multi-word identifiers, use capital letters to indicate the start of the word except the first word. For example, charArray, fileNumber, ClassName.

3. Avoid using underscores at the start of the identifier such as _read or _write.

4.5 Java Keywords

Keywords are predefined identifiers reserved by Java for a specific purpose. You cannot use keywords as names for your variables, classes, methods ...etc. Here is a list of the Java Keywords.

abstract	double	int	super
boolean	else	interface	switch
break	extends	long	synchronized
byte	false	native	this
byvalue	final	new	threadsafe
case	finally	null	throw
catch	float	package	transient
char	for	private	true
class	goto	protected	try
const	if	public	void
continue	implements	return	while
default	import	short	
do	instanceof	static	

We will try to discuss all the meanings of these keywords and how they are used in our Java programs as we go along the way.

4.6 Java Literals

Literals are tokens that do not change or are constant. The different types of literals in Java are: Integer Literals, Floating-Point Literals, Boolean Literals, Character Literals and String Literals.

4.6.1 Integer Literals

Integer literals come in different formats: **decimal** (base 10), **hexadecimal** (base 16), and **octal** (base 8). In using integer literals in our program, we have to follow some special notations.

For decimal numbers, we have no special notations. We just write a decimal number as it is. For hexadecimal numbers, it should be preceded by "0x" or "0X". For octals, they are preceded by "0".

For example, consider the number **12**. It's decimal representation is **12**, while in hexadecimal, it is **0xC**, and in octal, it is equivalent to **014**.

Integer literals default to the data type **int**. An int is a signed 32-bit value. In some cases, you may wish to force integer literal to the data type **long** by appending the "l" or "L" character. A long is a signed 64-bit value. We will cover more on data types later.

4.6.2 Floating-Point Literals

Floating point literals represent decimals with fractional parts. An example is 3.1415. Floating point literals can be expressed in standard or scientific notations. For example, 583.45 is in standard notation, while 5.8345e2 is in scientific notation.

Floating point literals default to the data type **double** which is a 64-bit value. To use a smaller precision (32-bit) **float**, just append the "f" or "F" character.

4.6.3 Boolean Literals

Boolean literals have only two values, **true** or **false**.

4.6.4 Character Literals

Character Literals represent single Unicode characters. A Unicode character is a 16-bit character set that replaces the 8-bit ASCII character set. Unicode allows the inclusion of symbols and special characters from other languages.

To use a character literal, enclose the character in single quote delimiters. For example, the letter **a**, is represented as **'a'**.

To use special characters such as a newline character, a backslash is used followed by the character code. For example, **'\n'** for the newline character, **'\r'** for the carriage return, **'\b'** for backspace.

4.6.5 String Literals

String literals represent multiple characters and are enclosed by double quotes. An example of a string literal is, **"Hello World"**.

4.7 Primitive data types

The Java programming language defines eight primitive data types. The following are, boolean (for logical), char (for textual), byte, short, int, long (integral), double and float (floating point).

4.7.1 Logical - boolean

A boolean data type represents two states: true and false. An example is,

```
boolean result = true;
```

The example shown above, declares a variable named **result** as **boolean** type and assigns it a value of **true**.

4.7.2 Textual - char

A character data type (char), represents a single Unicode character. It must have its literal enclosed in single quotes(' '). For example,

```
'a'    //The letter a
'\t'   //A tab
```

To represent special characters like ' (single quotes) or " (double quotes), use the escape character \. For example,

```
'\''   //for single quotes
'\\"'   //for double quotes
```

Although, String is not a primitive data type (it is a Class), we will just introduce String in this section. A String represents a data type that contains multiple characters. It is **not a primitive data type, it is a class**. It has its literal enclosed in double quotes("").

For example,

```
String message="Hello world!"
```

4.7.3 Integral – byte, short, int & long

Integral data types in Java uses three forms – decimal, octal or hexadecimal. Examples are,

```
2          //The decimal value 2
077        //The leading 0 indicates an octal value
0xBACC     //The leading 0x indicates a hexadecimal value
```

Integral types has `int` as default data type. You can define its `long` value by appending the letter `l` or `L`. Integral data type have the following ranges:

Integer Length	Name or Type	Range
8 bits	byte	-2^7 to 2^7-1
16 bits	short	-2^{15} to $2^{15}-1$
32 bits	int	-2^{31} to $2^{31}-1$
64 bits	long	-2^{63} to $2^{63}-1$

Table 8: Integral types and their ranges

Coding Guidelines:

In defining a long value, a lowercase L is not recommended because it is hard to distinguish from the digit 1.

4.7.4 Floating Point – float and double

Floating point types has double as default data type. Floating-point literal includes either a decimal point or one of the following,

```
E or e //(add exponential value)
F or f //(float)
D or d //(double)
```

Examples are,

```
3.14           //A simple floating-point value (a double)
6.02E23        //A large floating-point value
2.718F         //A simple float size value
123.4E+306D    //A large double value with redundant D
```

In the example shown above, the 23 after the E in the second example is implicitly positive. That example is equivalent to 6.02E+23. Floating-point data types have the following ranges:

Float Length	Name or Type	Range
32 bits	float	-2^{31} to $2^{31}-1$
64 bits	double	-2^{63} to $2^{63}-1$

Table 9: Floating point types and their ranges

4.8 Variables

A **variable** is an item of data used to store state of objects.

A variable has a **data type** and a **name**. The **data type** indicates the type of value that the variable can hold. The **variable name** must follow rules for identifiers.

4.8.1 Declaring and Initializing Variables

To declare a variable is as follows,

```
<data type> <name> [=initial value];
```

Note: Values enclosed in <> are required values, while those values enclosed in [] are optional.

Here is a sample program that declares and initializes some variables,

```
public class VariableSamples
{
    public static void main( String[] args ){
        //declare a data type with variable name
        // result and boolean data type
        boolean    result;

        //declare a data type with variable name
        // option and char data type
        char        option;
        option = 'C';    //assign 'C' to option

        //declare a data type with variable name
        //grade, double data type and initialized
        //to 0.0
        double    grade = 0.0;
    }
}
```

Coding Guidelines:

1. It always good to **initialize** your variables as you declare them.
2. Use **descriptive** names for your variables. Like for example, if you want to have a variable that contains a grade for a student, name it as, **grade** and not just some random letters you choose.

3. Declare one variable per line of code. For example, the variable declarations,

```
double exam=0;
double quiz=10;
double grade = 0;
```

is preferred over the declaration,

```
double exam=0, quiz=10, grade=0;
```

4.8.2 Outputting Variable Data

In order to output the value of a certain variable, we can use the following commands,

```
System.out.println()  
System.out.print()
```

Here's a sample program,

```
public class OutputVariable  
{  
    public static void main( String[] args ){  
        int value = 10;  
        char x;  
        x = 'A';  
  
        System.out.println( value );  
        System.out.println( "The value of x=" + x );  
    }  
}
```

The program will output the following text on screen,

```
10  
The value of x=A
```

4.8.3 System.out.println() vs. System.out.print()

What is the difference between the commands `System.out.println()` and `System.out.print()`? The first one appends a newline at the end of the data to output, while the latter doesn't.

Consider the statements,

```
System.out.print("Hello ");  
System.out.print("world!");
```

These statements will output the following on the screen,

```
Hello world!
```

Now consider the following statements,

```
System.out.println("Hello ");  
System.out.println("world!");
```

These statements will output the following on the screen,

```
Hello  
world!
```

4.8.4 Reference Variables vs. Primitive Variables

We will now differentiate the two types of variables that Java programs have. These are **reference variables** and **primitive variables**.

Primitive variables are variables with primitive data types. They store data in the actual memory location of where the variable is.


Reference variables are variables that stores the address in the memory location. It points to another memory location of where the actual data is. When you declare a variable of a certain **class**, you are actually declaring a reference variable to the object with that certain class.

For example, suppose we have two variables with data types int and String.

```
int num = 10;
String name = "Hello"
```

Suppose, the illustration shown below is the actual memory of your computer, wherein you have the address of the memory cells, the variable name and the data they hold.

Memory Address	Variable Name	Data
1001	num	10
:		:
1563	name	Address(2000)
:		:
:		:
2000		"Hello"



As you can see, for the primitive variable num, the data is on the actual location of where the variable is. For the reference variable name, the variable just holds the address of where the actual data is.

4.9 Operators

In Java, there are different types of operators. There are arithmetic operators, relational operators, logical operators and conditional operators. These operators follow a certain kind of precedence so that the compiler will know which operator to evaluate first in case multiple operators are used in one statement.

4.9.1 Arithmetic operators

Here are the basic arithmetic operators that can be used in creating your Java programs,

Operator	Use	Description
+	op1 + op2	Adds op1 and op2
*	op1 * op2	Multiplies op1 by op2
/	op1 / op2	Divides op1 by op2
%	op1 % op2	Computes the remainder of dividing op1 by op2
-	op1 - op2	Subtracts op2 from op1

Table 10: Arithmetic operations and their functions

Here's a sample program in the usage of these operators:

```
public class ArithmeticDemo
{
    public static void main(String[] args)
    {
        //a few numbers
        int i = 37;
        int j = 42;
        double x = 27.475;
        double y = 7.22;
        System.out.println("Variable values...");
        System.out.println("    i = " + i);
        System.out.println("    j = " + j);
        System.out.println("    x = " + x);
        System.out.println("    y = " + y); //adding numbers
        System.out.println("Adding...");
        System.out.println("    i + j = " + (i + j));
        System.out.println("    x + y = " + (x + y));

        //subtracting numbers
        System.out.println("Subtracting...");
        System.out.println("    i - j = " + (i - j));
        System.out.println("    x - y = " + (x - y));

        //multiplying numbers
        System.out.println("Multiplying...");
        System.out.println("    i * j = " + (i * j));
        System.out.println("    x * y = " + (x * y));
        //dividing numbers
        System.out.println("Dividing...");
        System.out.println("    i / j = " + (i / j));
        System.out.println("    x / y = " + (x / y));

        //computing the remainder resulting from dividing
        numbers
        System.out.println("Computing the remainder...");
        System.out.println("    i % j = " + (i % j));
        System.out.println("    x % y = " + (x % y));

        //mixing types
        System.out.println("Mixing types...");
        System.out.println("    j + y = " + (j + y));
        System.out.println("    i * x = " + (i * x));
    }
}
```

Here is the output of the program,

```
Variable values...
  i = 37
  j = 42
  x = 27.475
  y = 7.22
Adding...
  i + j = 79
  x + y = 34.695
Subtracting...
  i - j = -5
  x - y = 20.255
Multiplying...
  i * j = 1554
  x * y = 198.37
Dividing...
  i / j = 0
  x / y = 3.8054
Computing the remainder...
  i % j = 37
  x % y = 5.815
Mixing types...
  j + y = 49.22
  i * x = 1016.58
```

Note: When an integer and a floating-point number are used as operands to a single arithmetic operation, the result is a floating point. The integer is implicitly converted to a floating-point number before the operation takes place.

4.9.1.1 Increment and Decrement operators

Aside from the basic arithmetic operators, Java also includes a unary increment operator (++) and unary decrement operator (--). Increment and decrement operators increase and decrease a value stored in a number variable by 1.

For example, the expression,

```
count = count + 1;           //increment the value of count by 1
```

is equivalent to,

```
count++;
```

Operator	Use	Description
++	op++	Increments op by 1; evaluates to the value of op before it was incremented
++	++op	Increments op by 1; evaluates to the value of op after it was incremented
--	op--	Decrements op by 1; evaluates to the value of op before it was decremented
--	--op	Decrements op by 1; evaluates to the value of op after it was decremented

Table 11: Increment and Decrement operators

The increment and decrement operators can be placed before or after an operand.

When used **before** an operand, it causes the variable to be incremented or decremented by 1, and then the new value is used in the expression in which it appears. For example,

```
int i = 10,  
int j = 3;  
int k = 0;  
  
k = ++j + i; //will result to k = 4+10 = 14
```


When the increment and decrement operators are placed after the operand, the old value of the variable will be used in the expression where it appears. For example,

```
int i = 10,  
int j = 3;  
int k = 0;  
  
k = j++ + i; //will result to k = 3+10 = 13
```

Coding Guideline:

Always keep expressions containing increment and decrement operators simple and easy to understand.

4.9.2 Relational operators

Relational operators compare two values and determines the relationship between those values. The output of evaluation are the **boolean values** true or false.

<i>Operator</i>	<i>Use</i>	<i>Description</i>
>	op1 > op2	op1 is greater than op2
>=	op1 >= op2	op1 is greater than or equal to op2
<	op1 < op2	op1 is less than op2
<=	op1 <= op2	op1 is less than or equal to op2
==	op1 == op2	op1 and op2 are equal
!=	op1 != op2	op1 and op2 are not equal

Table 12: Relational Operators

Here's a sample program that uses relational operators,

```
public class RelationalDemo
{
    public static void main(String[] args) {
        //a few numbers
        int i = 37;
        int j = 42;
        int k = 42;
        System.out.println("Variable values...");
        System.out.println("    i = " + i);
        System.out.println("    j = " + j);
        System.out.println("    k = " + k);

        //greater than
        System.out.println("Greater than...");
        System.out.println("    i > j = " + (i > j)); //false
        System.out.println("    j > i = " + (j > i)); //true
        System.out.println("    k > j = " + (k > j)); //false

        //greater than or equal to
        System.out.println("Greater than or equal to...");
        System.out.println("    i >= j = " + (i >= j)); //false
        System.out.println("    j >= i = " + (j >= i)); //true
        System.out.println("    k >= j = " + (k >= j)); //true

        //less than
        System.out.println("Less than...");
        System.out.println("    i < j = " + (i < j)); //true
        System.out.println("    j < i = " + (j < i)); //false
        System.out.println("    k < j = " + (k < j)); //false

        //less than or equal to
        System.out.println("Less than or equal to...");
        System.out.println("    i <= j = " + (i <= j)); //true
        System.out.println("    j <= i = " + (j <= i)); //false
        System.out.println("    k <= j = " + (k <= j)); //true

        //equal to
        System.out.println("Equal to...");
        System.out.println("    i == j = " + (i == j)); //false
        System.out.println("    k == j = " + (k == j)); //true

        //not equal to
        System.out.println("Not equal to...");
        System.out.println("    i != j = " + (i != j)); //true
        System.out.println("    k != j = " + (k != j)); //false
    }
}
```

Here's the output from this program:

```
Variable values...
  i = 37
  j = 42
  k = 42
Greater than...
  i > j = false
  j > i = true
  k > j = false
Greater than or equal to...
  i >= j = false
  j >= i = true
  k >= j = true
Less than...
  i < j = true
  j < i = false
  k < j = false
Less than or equal to...
  i <= j = true
  j <= i = false
  k <= j = true
Equal to...
  i == j = false
  k == j = true
Not equal to...
  i != j = true
  k != j = false
```

4.9.3 Logical operators

Logical operators have one or two boolean operands that yield a boolean result. There are six logical operators: `&&` (logical AND), `&` (boolean logical AND), `||` (logical OR), `|` (boolean logical inclusive OR), `^` (boolean logical exclusive OR), and `!` (logical NOT).

The basic expression for a logical operation is,

$$x1 \text{ op } x2$$

where `x1`, `x2` can be boolean expressions, variables or constants, and `op` is either `&&`, `&`, `||`, `|` or `^` operator. The truth tables that will be shown next, summarize the result of each operation for all possible combinations of `x1` and `x2`.

4.9.3.1 && (logical AND) and & (boolean logical AND)

Here is the truth table for && and &,

<i>x1</i>	<i>x2</i>	<i>Result</i>
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

Table 13: Truth table for & and &&

The basic difference between && and & operators is that && supports **short-circuit evaluations** (or partial evaluations), while & doesn't. What does this mean?

Given an expression,

```
exp1 && exp2
```

&& will evaluate the expression exp1, and immediately return a false value if exp1 is false. If exp1 is false, the operator never evaluates exp2 because the result of the operator will be false regardless of the value of exp2. In contrast, the & operator always evaluates both exp1 and exp2 before returning an answer.

Here's a sample source code that uses logical and boolean AND,

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

        int    i      = 0;
        int    j      = 10;
        boolean test= false;

        //demonstrate &&
        test = (i > 10) && (j++ > 9);
        System.out.println(i);
        System.out.println(j);
        System.out.println(test);

        //demonstrate &
        test = (i > 10) & (j++ > 9);
        System.out.println(i);
        System.out.println(j);
        System.out.println(test);
    }
}
```

The output of the program is,

```
0
10
false
0
11
false
```

Note, that the `j++` on the line containing the `&&` operator is not evaluated since the first expression `(i>10)` is already equal to `false`.

4.9.3.2 || (logical OR) and | (boolean logical inclusive OR)

Here is the truth table for || and |,

<i>x1</i>	<i>x2</i>	<i>Result</i>
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Table 14: Truth table for | and ||

The basic difference between || and | operators is that || supports short-circuit evaluations (or partial evaluations), while | doesn't. What does this mean?

Given an expression,

`exp1 || exp2`

|| will evaluate the expression exp1, and immediately return a true value if exp1 is true. If exp1 is true, the operator never evaluates exp2 because the result of the operator will be true regardless of the value of exp2. In contrast, the | operator always evaluates both exp1 and exp2 before returning an answer.

Here's a sample source code that uses logical and boolean OR,

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

        int    i      = 0;
        int    j      = 10;
        boolean test= false;

        //demonstrate ||
        test = (i < 10) || (j++ > 9);
        System.out.println(i);
        System.out.println(j);
        System.out.println(test);

        //demonstrate |
        test = (i < 10) | (j++ > 9);
        System.out.println(i);
        System.out.println(j);
        System.out.println(test);
    }
}
```


The output of the program is,

```
0
10
true
0
11
true
```

Note, that the `j++` on the line containing the `||` operator is not evaluated since the first expression (`i<10`) is already equal to `true`.

4.9.3.3 ^ (boolean logical exclusive OR)

Here is the truth table for ^,

<i>x1</i>	<i>x2</i>	<i>Result</i>
TRUE	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Table 15: Truth table for ^

The result of an exclusive OR operation is TRUE, if and only if one operand is true and the other is false. Note that both operands must always be evaluated in order to calculate the result of an exclusive OR.

Here's a sample source code that uses the logical exclusive OR operator,

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

        boolean val1 = true;
        boolean val2 = true;
        System.out.println(val1 ^ val2);

        val1 = false;
        val2 = true;
        System.out.println(val1 ^ val2);

        val1 = false;
        val2 = false;
        System.out.println(val1 ^ val2);

        val1 = true;
        val2 = false;
        System.out.println(val1 ^ val2);
    }
}
```

The output of the program is,

```
false
true
false
true
```

4.9.3.4 ! (logical NOT)

The logical NOT takes in one argument, wherein that argument can be an expression, variable or constant. Here is the truth table for !,

<i>x1</i>	<i>Result</i>
TRUE	FALSE
FALSE	TRUE

Table 16: Truth table for !

Here's a sample source code that uses the logical NOT operator,

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

        boolean val1 = true;
        boolean val2 = false;
        System.out.println(!val1);
        System.out.println(!val2);

    }
}
```

The output of the program is,

```
false
true
```

4.9.4 Conditional Operator (?:)

The conditional operator **?:** is a ternary operator. This means that it takes in three arguments that together form a conditional expression. The structure of an expression using a conditional operator is,

`exp1?exp2:exp3`

wherein `exp1` is a boolean expression whose result must either be true or false.

If `exp1` is true, `exp2` is the value returned. If it is false, then `exp3` is returned.

For example, given the code,

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

        String      status = "";
        int    grade = 80;

        //get status of the student
        status = (grade >= 60)?"Passed":"Fail";

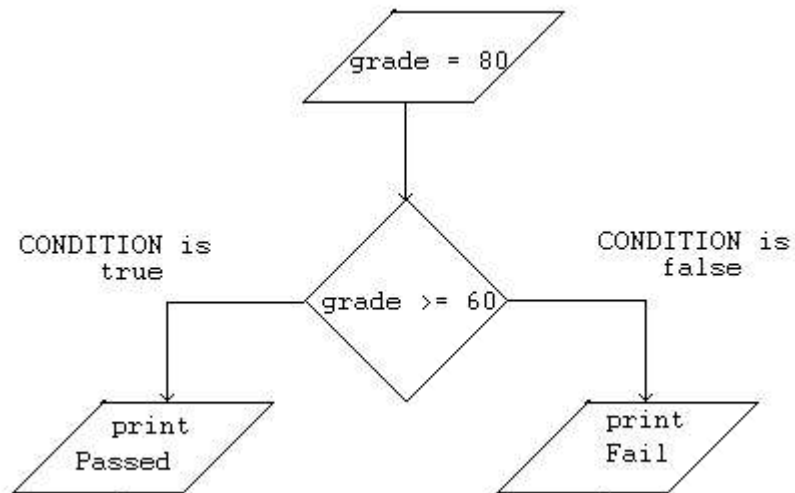
        //print status
        System.out.println( status );

    }
}
```

The output of this program will be,

Passed

Here is the flowchart of how `?:` works,



Here is another program that uses the `?:` operator,

```
class ConditionalOperator
{
    public static void main( String[] args ){
        int    score = 0;
        char   answer = 'a';

        score = (answer == 'a') ? 10 : 0;
        System.out.println("Score = " + score );
    }
}
```

The output of the program is,

```
Score = 10
```

4.9.5 Operator Precedence

Operator precedence defines the compiler's order of evaluation of operators so as to come up with an unambiguous result.

The following is the list of Java operators from highest to lowest precedence:

.	[]	()	
++	--	!	~
*	/	%	
+	-		
<<	>>	>>>	<<<
<	>	<=	>=
==	!=		
&			
^			
&&			
?:			
=			

The highest precedence is on the top row and the lowest is on the bottom row.

Given a complicated expression,

$$6\%2*5+4/2+88-10$$

we can re-write the expression and place some parenthesis base on operator precedence,

$$((6\%2)*5)+(4/2)+88-10;$$

Coding Guidelines

To avoid confusion in evaluating mathematical operations, keep your expressions simple and use parenthesis.

4.10 Exercises

4.10.1 Declaring and printing variables

Given the table below, declare the following variables with the corresponding data types and initialization values. Output to the screen the variable names together with the values.

Variable name	Data Type	Initial value
number	integer	10
letter	character	a
result	boolean	true
str	String	hello

The following should be the expected screen output,

```
Number = 10
letter = a
result = true
str = hello
```

4.10.2 Getting the average of three numbers

Create a program that outputs the average of three numbers. Let the values of the three numbers be, 10, 20 and 45. The expected screen output is,

```
number 1 = 10
number 2 = 20
number 3 = 45
Average is = 25
```

4.10.3 Output greatest value

Given three numbers, write a program that outputs the number with the greatest value among the three. Use the conditional `?:` operator that we have studied so far (**HINT**: You will need to use two sets of `?:` to solve this). For example, given the numbers 10, 23 and 5, your program should output,

```
number 1 = 10
number 2 = 23
number 3 = 5
The highest number is = 23
```

4.10.4 Operator precedence

Given the following expressions, re-write them by writing some parenthesis based on the sequence on how they will be evaluated.

1. $a / b ^ c ^ d - e + f - g * h + i$
2. $3 * 10 * 2 / 15 - 2 + 4 ^ 2 ^ 2$
3. $r ^ s * t / u - v + w ^ x - y ++$

5 Getting Input from the Keyboard

Now that we've studied some basic concepts in Java and we've written some simple programs, let's make our programs more interactive by getting some input from the user. In this section, we'll be discussing two methods of getting input, the first one is through the use of the `BufferedReader` class and the other one involves a graphical user interface by using `JOptionPane`.

5.1 Using *BufferedReader* to get input

In this section, we will use the `BufferedReader` class found in the `java.io` package in order to get input from the keyboard.

Here are the steps to get input from the keyboard:

1. Add this at the top of your code:

```
import java.io.*;
```

2. Add this statement:

```
BufferedReader dataIn = new BufferedReader(  
    new InputStreamReader( System.in) );
```

3. Declare a temporary `String` variable to get the input, and invoke the `readLine()` method to get input from the keyboard. You have to type it inside a try-catch block.

```
try{  
    String temp = dataIn.readLine();  
}  
catch( IOException e ){  
    System.out.println("Error in getting input");  
}
```


Here is the complete source code:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class GetInputFromKeyboard
{
    public static void main( String[] args ){

        BufferedReader dataIn = new BufferedReader(new
            InputStreamReader( System.in) );

        String name = "";

        System.out.print("Please Enter Your Name:");

        try{
            name = dataIn.readLine();
        }catch( IOException e ){
            System.out.println("Error!");
        }

        System.out.println("Hello " + name + "!");
    }
}
```

Now let's try to explain each line of code:

The statements,

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
```

indicate that we want to use the classes **BufferedReader**, **InputStreamReader** and **IOException** which is inside the **java.io package**. The Java Application Programming Interface (API) contains hundreds of predefined classes that you can use in your programs. These classes are organized into what we call **packages**.

Packages contain classes that have related purpose. Just like in our example, the **java.io** package contains classes that allow programs to input and output data. The statements can also be rewritten as,

```
import java.io.*;
```

which will load all the classes found in the package, and then we can use those classes inside our program.

The next two statements,

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

were already discussed in the previous lesson. This means we declare a class named `GetInputFromKeyboard` and we declare the `main` method.

In the statement,

```
BufferedReader dataIn = new BufferedReader(new
    InputStreamReader( System.in) );
```

we are declaring a variable named **dataIn** with the class type **BufferedReader**. Don't worry about what the syntax means for now. We will cover more about this later in the course.

Now, we are declaring a `String` variable with the identifier `name`,

```
String name = "";
```

This is where we will store the input of the user. The variable `name` is initialized to an empty `String` `""`. It is always good to initialize your variables as you declare them.

The next line just outputs a `String` on the screen asking for the user's name.

```
System.out.print("Please Enter Your Name:");
```

Now, the following block defines a try-catch block,

```
try{
    name = dataIn.readLine();
}catch( IOException e ){
    System.out.println("Error!");
}
```

This assures that the possible exceptions that could occur in the statement

```
name = dataIn.readLine();
```

will be caught. We will cover more about exception handling in the latter part of this course, but for now, just take note that you need to add this code in order to use the `readLine()` method of `BufferedReader` to get input from the user.

Now going back to the statement,

```
name = dataIn.readLine();
```

the method call, `dataIn.readLine()`, gets input from the user and will return a `String` value. This value will then be saved to our `name` variable, which we will use in our final statement to greet the user,

```
System.out.println("Hello " + name + "!");
```

5.2 Using JOptionPane to get input

Another way to get input from the user is by using the `JOptionPane` class which is found in the `javax.swing` package. `JOptionPane` makes it easy to pop up a standard dialog box that prompts users for a value or informs them of something.

Given the following code,

```
import javax.swing.JOptionPane;

public class GetInputFromKeyboard
{
    public static void main( String[] args ){
        String name = "";
        name = JOptionPane.showInputDialog("Please enter your
                                         name");

        String msg = "Hello " + name + "!";

        JOptionPane.showMessageDialog(null, msg);
    }
}
```

This will output,



The first statement,

```
import javax.swing.JOptionPane;
```

indicates that we want to import the class `JOptionPane` from the `javax.swing` package.

We can also write this as,

```
import javax.swing.*;
```

The statement,

```
name = JOptionPane.showInputDialog("Please enter your name");
```

creates a `JOptionPane` input dialog, which will display a dialog with a message, a textfield and an OK button as shown in the figure. This returns a `String` which we will save in the `name` variable.

Now we create the welcome message, which we will store in the `msg` variable,

```
String msg = "Hello " + name + "!";
```

The next line displays a dialog which contains a message and an OK button.

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

5.3 Exercises

5.3.1 Last 3 words (*BufferedReader version*)

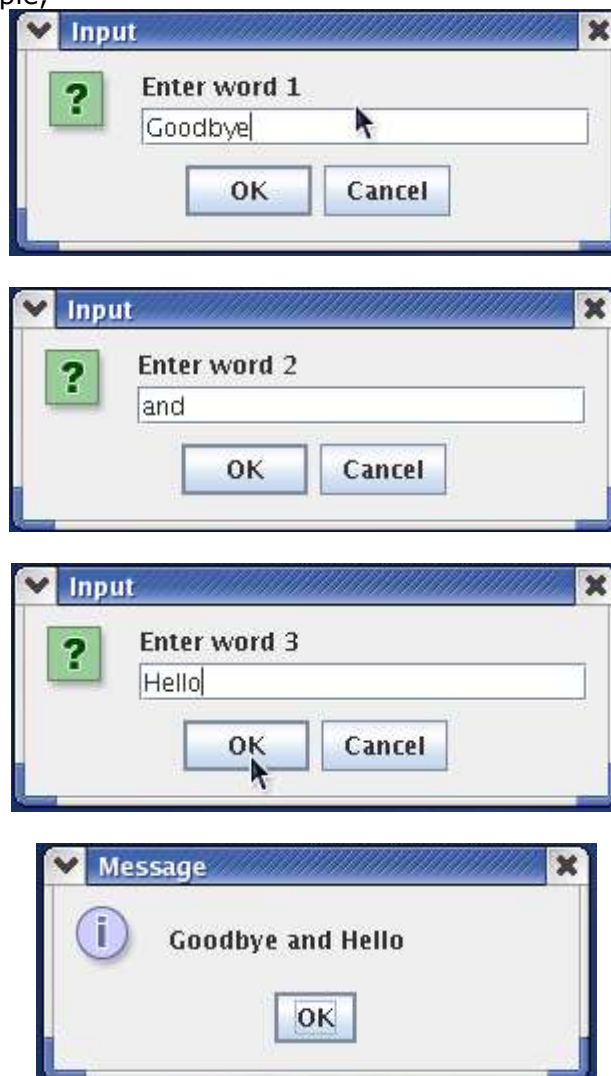
Using `BufferedReader`, ask for three words from the user and output those three words on the screen. For example,

```
Enter word1:Goodbye
Enter word2:and
Enter word3:Hello

Goodbye and Hello
```

5.3.2 Last 3 words (*JOptionPane version*)

Using `JOptionPane`, ask for three words from the user and output those three words on the screen. For example,



6 Control Structures

In the previous sections, we have given examples of sequential programs, wherein statements are executed one after another in a fixed order. In this section, we will be discussing control structures, which allows us to change the ordering of how the statements in our programs are executed.

There are two types of control structures. The first one is decision control structures, which allows us to select specific sections of code to be executed. The other one is repetition control structures, which allows us to execute specific sections of the code a number of times.

6.1 Decision Control Structures

Decision control structures are Java statements that allows us to select and execute specific blocks of code while skipping other sections.

6.1.1 *if* statement

The if-statement specifies that a statement (or block of code) will be executed *if and only if a certain boolean statement is true*.

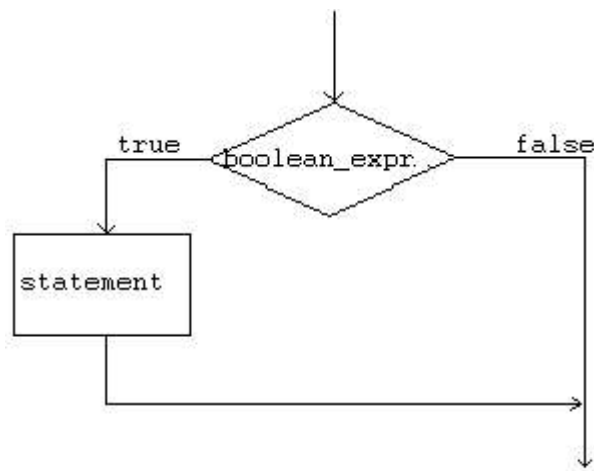
The if-statement has the form,

```
if( boolean_expression )
    statement;
```

or

```
if( boolean_expression ){
    statement1;
    statement2;
    . . .
}
```

where, `boolean_expression` is either a boolean expression or boolean variable.



For example, given the code snippet,

```
int grade = 68;  
  
if( grade > 60 ) System.out.println("Congratulations!");
```

or

```
int grade = 68;  
  
if( grade > 60 ){  
    System.out.println("Congratulations!");  
    System.out.println("You passed!");  
}
```

Coding Guidelines:

1. The **boolean_expression** part of a statement should evaluate to a boolean value. That means that the execution of the condition should either result to a value of **true** or a **false**.
2. Indent the statements inside the if-block. For example,

```
if( boolean_expression ){  
    //statement1;  
    //statement2;  
}
```


6.1.2 if-else statement

The if-else statement is used when we want to execute a certain statement if a condition is true, and a different statement if the condition is false.

The if-else statement has the form,

```
if( boolean_expression )
    statement;
else
    statement;
```

or can also be written as,

```
if( boolean_expression ){
    statement1;
    statement2;
    . . .
}
else{
    statement1;
    statement2;
    . . .
}
```

For example, given the code snippet,

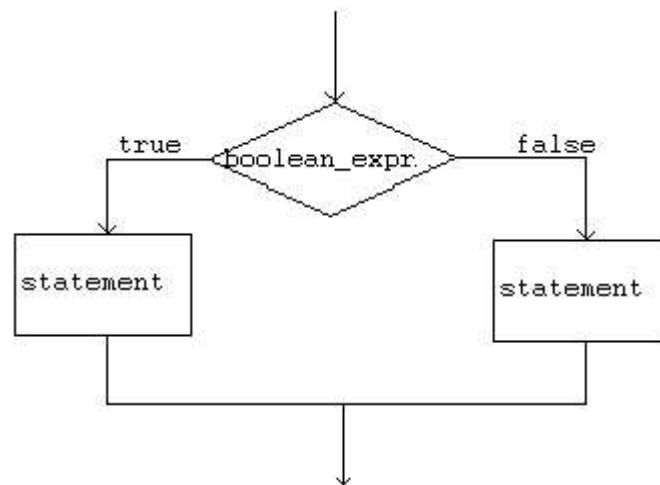
```
int grade = 68;

if( grade > 60 )    System.out.println("Congratulations!");
else                System.out.println("Sorry you failed");
```

or

```
int grade = 68;

if( grade > 60 ){
    System.out.println("Congratulations!");
    System.out.println("You passed!");
}
else{
    System.out.println("Sorry you failed");
}
```

**Coding Guidelines:**

1. To avoid confusion, always place the statement or statements of an if or if-else block inside brackets {}.
2. You can have nested if-else blocks. This means that you can have other if-else blocks inside another if-else block. For example,

```
if( boolean_expression ){  
    if( boolean_expression ){  
        . . .  
    }  
}  
else{ . . .  
}
```

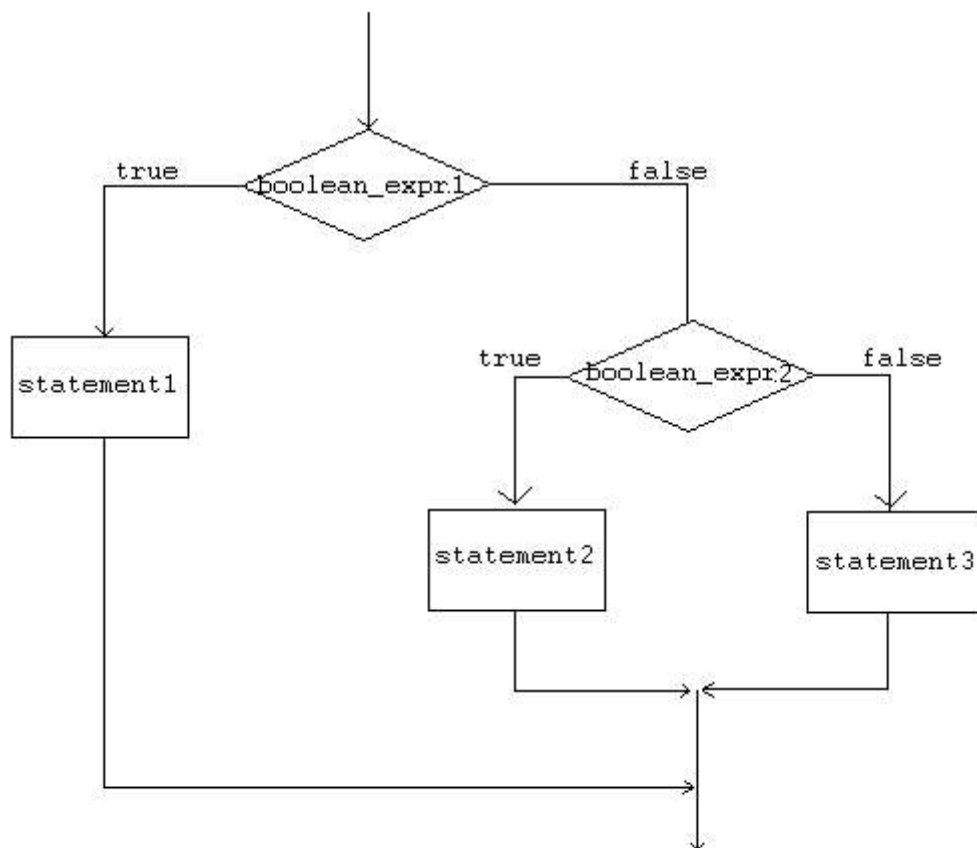
6.1.3 if-else-if statement

The statement in the else-clause of an if-else block can be another if-else structures. This cascading of structures allows us to make more complex selections.

The if-else if statement has the form,

```
if( boolean_expression1 )
    statement1;
else if( boolean_expression2 )
    statement2;
else
    statement3;
```

Take note that you can have many else-if blocks after an if-statement. The else-block is optional and can be omitted. In the example shown above, if `boolean_expression1` is true, then the program executes `statement1` and skips the other statements. If `boolean_expression2` is true, then the program executes `statement2` and skips to the statements following `statement3`.



For example, given the code snippet,

```
int grade = 68;

if( grade > 90 ){
    System.out.println("Very good!");
}
else if( grade > 60 ){
    System.out.println("Very good!");
}
else{
    System.out.println("Sorry you failed");
}
```

6.1.4 Common Errors when using the if-else statements:

1. The condition inside the if-statement does not evaluate to a boolean value. For example,

```
//WRONG
int number = 0;
if( number ){
    //some statements here
}
```

The variable number does not hold a Boolean value.

2. Using = instead of == for comparison. For example,

```
//WRONG
int number = 0;
if( number = 0 ){
    //some statements here
}
```

This should be written as,

```
//CORRECT
int number = 0;
if( number == 0 ){
    //some statements here
}
```

3. Writing **elseif** instead of **else if**.

6.1.5 Example for if-else-else if

```
public class Grade
{
    public static void main( String[] args )
    {
        double grade = 92.0;

        if( grade >= 90 ){
            System.out.println( "Excellent!" );
        }
        else if( (grade < 90) && (grade >= 80)){
            System.out.println("Good job!" );
        }
        else if( (grade < 80) && (grade >= 60)){
            System.out.println("Study harder!" );
        }
        else{
            System.out.println("Sorry, you failed.");
        }
    }
}
```

6.1.6 *switch statement*

Another way to indicate a branch is through the **switch** keyword. The switch construct allows branching on multiple outcomes.

The switch statement has the form,

```
switch( switch_expression ){
    case case_selector1:
        statement1; //
        statement2; //block 1
        . . .      //
        break;

    case case_selector2:
        statement1; //
        statement2; //block 2
        . . .      //
        break;

    . . .
    default:
        statement1; //
        statement2; //block n
        . . .      //
        break;
}
```

where, `switch_expression` is an **integer** or **character** expression and, `case_selector1`, `case_selector2` and so on, are unique integer or character constants.

When a switch is encountered, Java first evaluates the `switch_expression`, and jumps to the case whose selector matches the value of the expression. The program executes the statements in order from that point on until a `break` statement is encountered, skipping then to the first statement after the end of the switch structure.

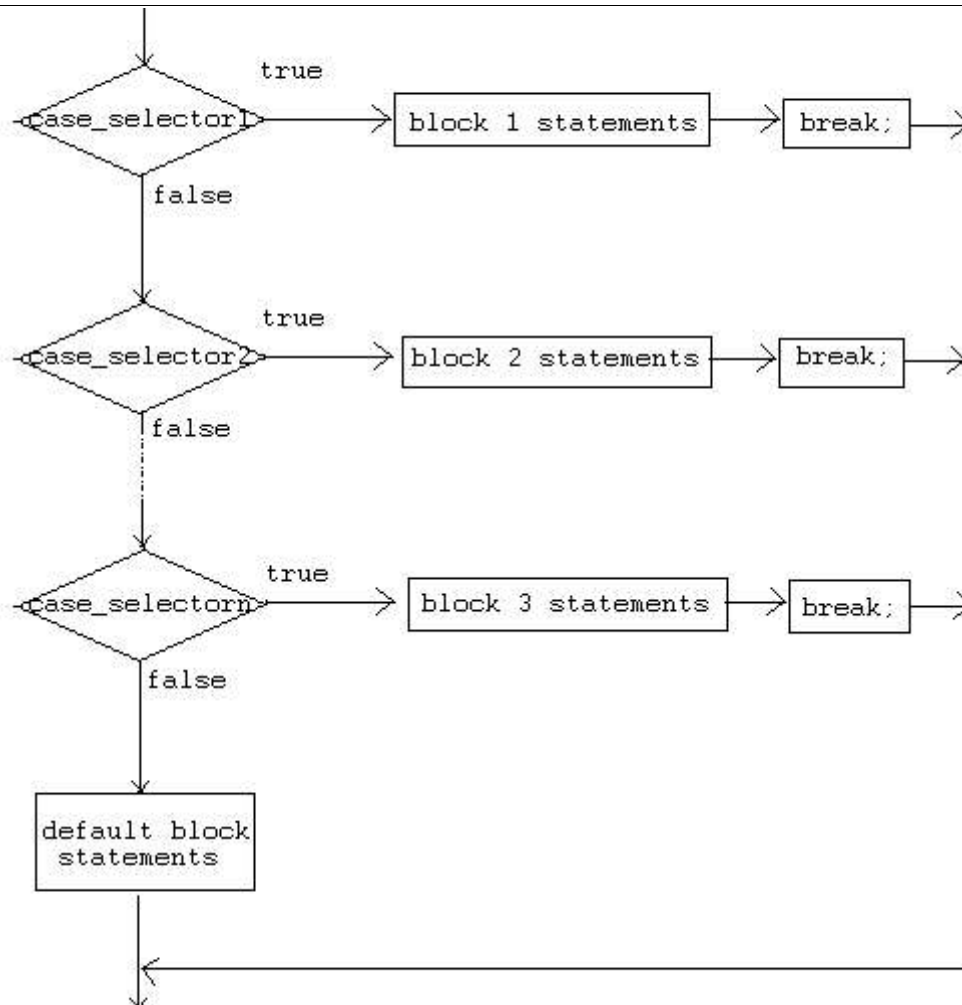
If none of the cases are satisfied, the default block is executed. Take note however, that the default part is optional. A switch statement can have no default block.

NOTE:

- Unlike with the **if** statement, the multiple statements are executed in the switch statement without needing the curly braces.
- When a case in a switch statement has been matched, all the statements associated with that case are executed. Not only that, the statements associated with the succeeding cases are also executed.
- To prevent the program from executing statements in the subsequent cases, we use a **break** statement as our last statement.

Coding Guidelines:

1. Deciding whether to use an if statement or a switch statement is a judgment call. You can decide which to use, based on readability and other factors.
2. An if statement can be used to make decisions based on ranges of values or conditions, whereas a switch statement can make decisions based only on a single integer or character value. Also, the value provided to each case statement must be unique.



6.1.7 Example for switch

```
public class Grade
{
    public static void main( String[] args )
    {
        double grade = 92.0;

        switch(grade){
        case 100:
            System.out.println( "Excellent!" );
            break;
        case 90:
            System.out.println("Good job!" );
            break;
        case 80:
            System.out.println("Study harder!" );
            break;
        default:
            System.out.println("Sorry, you failed.");
        }
    }
}
```


6.2 Repetition Control Structures

Repetition control structures are Java statements that allows us to execute specific blocks of code a number of times. There are three types of repetition control structures, the while, do-while and for loops.

6.2.1 while loop

The while loop is a statement or block of statements that is repeated as long as some condition is satisfied.

The while statement has the form,

```
while( boolean_expression ){
    statement1;
    statement2;
    . . .
}
```

The statements inside the while loop are executed as long as the boolean_expression evaluates to true.

For example, given the code snippet,

```
int i = 4;
while ( i > 0 ){
    System.out.print(i);
    i--;
}
```

The sample code shown will print 4321 on the screen. Take note that if the line containing the statement `i--;` is removed, this will result to an **infinite loop**, or a loop that does not terminate. Therefore, when using while loops or any kind of repetition control structures, make sure that you add some statements that will allow your loop to terminate at some point.

The following are other examples of while loops,

Example 1:

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

Example 2:

```
//infinite loop
while(true)
    System.out.println("hello");
```

Example 3:

```
//no loops
// statement is not even executed
while (false)
    System.out.println("hello");
```

6.2.2 *do-while loop*

The do-while loop is similar to the while-loop. The statements inside a do-while loop are executed several times as long as the condition is satisfied.

The main difference between a while and do-while loop is that, the statements inside a do-while loop are executed **at least once**.

The do-while statement has the form,

```
do{
    statement1;
    statement2;
    . . .
}while( boolean_expression );
```

The statements inside the do-while loop are first executed, and then the condition in the `boolean_expression` part is evaluated. If this evaluates to true, the statements inside the do-while loop are executed again.

Here are a few examples that uses the do-while loop:

Example 1:

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

This example will output 0123456789 on the screen.

Example 2:

```
//infinite loop
do{
    System.out.println("hello");
} while (true);
```

This example will result to an infinite loop, that prints `hello` on screen.

Example 3:

```
//one loop
// statement is executed once
do
    System.out.println("hello");
while (false);
```

This example will output `hello` on the screen.

Coding Guidelines:

1. Common programming mistakes when using the do-while loop is forgetting to write the semi-colon after the while expression.
`do{
 ...
}while(boolean_expression) //WRONG->forgot semicolon ;`
2. Just like in while loops, make sure that your do-while loops will terminate at some point.

6.2.3 for loop

The for loop, like the previous loops, allows execution of the same code a number of times.

The for loop has the form,

```
for (InitializationExpression; LoopCondition; StepExpression){  
    statement1;  
    statement2;  
    . . .  
}
```

where,

InitializationExpression -initializes the loop variable.

LoopCondition - compares the loop variable to some limit value.

StepExpression - updates the loop variable.

A simple example of the for loop is,

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

In this example, the statement `i=0`, first initializes our variable. After that, the condition expression `i<10` is evaluated. If this evaluates to true, then the statement inside the for loop is executed. Next, the expression `i++` is executed, and then the condition expression is again evaluated. This goes on and on, until the condition expression evaluates to false.

This example, is equivalent to the while loop shown below,

```
int i = 0;  
while( i < 10 ){  
    System.out.print(i);  
    i++;  
}
```

6.3 Exercises

6.3.1 Grades

Get three exam grades from the user and compute the average of the grades. Output the average of the three exams. Together with the average, also include a smiley face in the output if the average is greater than or equal to 60, otherwise output :-(.

1. Use `BufferedReader` to get input from the user, and `System.out` to output the result.
2. Use `JOptionPane` to get input from the user and to output the result.

6.3.2 Number in words

Get a number as input from the user, and output the equivalent of the number in words. The number inputted should range from 1-10. If the user inputs a number that is not in the range, output, "Invalid number".

1. Use an if-else statement to solve this problem
2. Use a switch statement to solve this problem

6.3.3 Hundred Times

Create a program that prints your name a hundred times. Do three versions of this program using a while loop, a do-while loop and a for-loop.

6.3.4 Powers

Compute the power of a number given the base and exponent. Do three versions of this program using a while loop, a do-while loop and a for-loop.

7 Java Arrays

In this section, we will be discussing about Java Arrays. First, we are going to define what arrays are, and then we are going to discuss on how to declare and use them.

7.1 Introduction to arrays

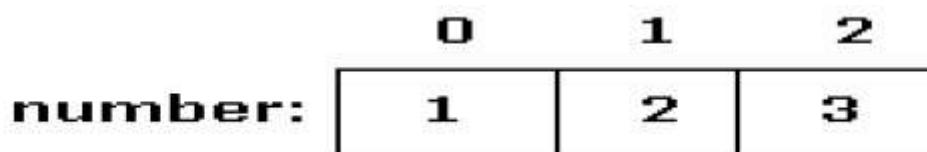
In the previous sections, we have discussed on how to declare different variables using the primitive data types. In declaring variables, we often use a unique identifier or name and a datatype. In order to use the variable, we call it by its identifier name.

For example, we have here three variables of type `int` with different identifiers for each variable.

```
int number1;  
int number2;  
int number3;  
  
number1 = 1;  
number2 = 2;  
number3 = 3;
```

As you can see, it seems like a tedious task in order to just initialize and use the variables especially if they are used for the same purpose. In Java and other programming languages, there is one capability wherein we can use one variable to store a list of data and manipulate them more efficiently. This type of variable is called an **array**.

An **array** stores *multiple data items of the same datatype*, in a contiguous block of memory, divided into a number of slots. Think of an array as a stretched variable – a location that still has one identifier name, but can hold more than one value.



7.2 Declaring Arrays

Arrays must be declared like all variables. When declaring an array, you list the data type, followed by a set of square brackets[], followed by the identifier name. For example,

```
int []ages;
```

or you can place the brackets after the identifier. For example,

```
int ages[];
```

After declaring, we must create the array and specify its length with a **constructor** statement. This process in Java is called **instantiation** (the Java word for creates). In order to instantiate an object, we need to use a **constructor** for this. We will cover more about instantiating objects and constructors later. Take note, that the size of an array cannot be changed once you've initialized it. For example,

```
//declaration
int ages[];

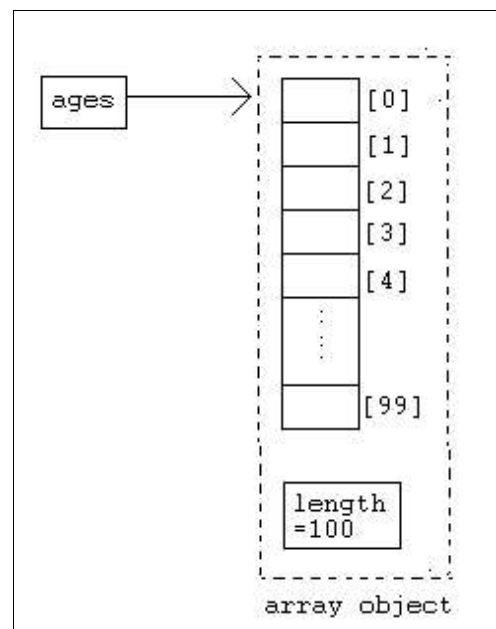
//instantiate object
ages = new int[100];
```

or, can also be written as,

```
//declare and instantiate object
int ages[] = new int[100];
```

In the example, the declaration tells the Java Compiler that the identifier `ages` will be used as the name of an array containing integers, and to create or instantiate a new array containing 100 elements.

Instead of using the `new` keyword to instantiate an array, you can also automatically declare, construct and assign values at once.



Examples are,

```
//creates an array of boolean variables with identifier
//results. This array contains 4 elements that are
//initialized to values {true, false, true, false}
boolean results[] = { true, false, true, false };

//creates an array of 4 double variables initialized
//to the values {100, 90, 80, 75};
double []grades = {100, 90, 80, 75};

//creates an array of Strings with identifier days and
//initialized. This array contains 7 elements
String days[] = { "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"};
```


7.3 Accessing an array element

To access an array element, or a part of the array, you use a number called an **index** or a subscript.

An **index number or subscript** is assigned to each member of the array, allowing the program and the programmer to access individual values when necessary. Index numbers **are always integers**. They **begin with zero and progress sequentially by whole numbers to the end of the array**. Take note that the elements inside your array is from **0 to (sizeofArray-1)**.

For example, given the array we declared a while ago, we have

```
//assigns 10 to the first element in the array
ages[0] = 10;

//prints the last element in the array
System.out.print(ages[99]);
```

Take note that once an array is declared and constructed, the stored value of each member of the array will be initialized to zero for number data. However, reference data types such as Strings are not initialized to blanks or an empty string "". Therefore, you must populate the String arrays explicitly.

The following is a sample code on how to print all the elements in the array. This uses a for loop, so our code is shorter.

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

        int[] ages = new int[100];

        for( int i=0; i<100; i++ ){
            System.out.print( ages[i] );
        }
    }
}
```

Coding Guidelines:

1. *It is usually better to initialize or instantiate the array right away after you declare it. For example, the declaration,
 int []arr = new int[100];
is preferred over,
 int []arr;
 arr = new int[100];*
2. *The elements of an n-element array have indexes from 0 to n-1. Note that there is no array element arr[n]! This will result in an array-index-out-of-bounds exception.*

7.4 Array length

In order to get the number of elements in an array, you can use the **length** field of an array. The length field of an array returns the size of the array. It can be used by writing,

```
arrayName.length
```

For example, given the previous example, we can re-write it as,

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

        int[] ages = new int[100];

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

Coding Guidelines:

1. When creating for loops to process the elements of an array, use the array object's **length** field in the condition statement of the for loop. This will allow the loop to adjust automatically for different-sized arrays.
2. Declare the sizes of arrays in a Java program using named constants to make them easy to change. For example,

```
final int ARRAY_SIZE = 1000;    //declare a constant
...
int[] ages = new int[ARRAY_SIZE];
```

7.5 Multidimensional Arrays

Multidimensional arrays are implemented as arrays of arrays. Multidimensional arrays are declared by appending the appropriate number of bracket pairs after the array name. For example,

```
// integer array 512 x 128 elements
int[][] twoD = new int[512][128];

// character array 8 x 16 x 24
char[][][] threeD = new char[8][16][24];

// String array 4 rows x 2 columns
String[][] dogs = {{ "terry", "brown" },
                   { "Kristin", "white" },
                   { "toby", "gray"},
                   { "fido", "black"}
                  };
```

To access an element in a multidimensional array is just the same as accessing the elements in a one dimensional array. For example, to access the first element in the first row of the array dogs, we write,

```
System.out.print( dogs[0][0] );
```

This will print the String "terry" on the screen.

7.6 Exercises

7.6.1 Days of the Week

Create an array of Strings which are initialized to the 7 days of the week. For Example,

```
String days[] = {"Monday", "Tuesday"...};
```

Using a while-loop, print all the contents of the array. (do the same for do-while and for-loop)

7.6.2 Greatest number

Using BufferedReader or JOptionPane, ask for 10 numbers from the user. Use an array to store the values of these 10 numbers. Output on the screen the number with the greatest value.

7.6.3 Addressbook Entries

Given the following multidimensional array that contains addressbook entries:

```
String entry = {"Florence", "735-1234", "Manila"},  
               {"Joyce", "983-3333", "Quezon City"},  
               {"Becca", "456-3322", "Manila"};
```

Print the following entries on screen in the following format:

```
Name   : Florence  
Tel. #   : 735-1234  
Address  : Manila  
  
Name   : Joyce  
Tel. #   : 983-3333  
Address  : Quezon City  
  
Name   : Becca  
Tel. #   : 456-3322  
Address  : Manila
```

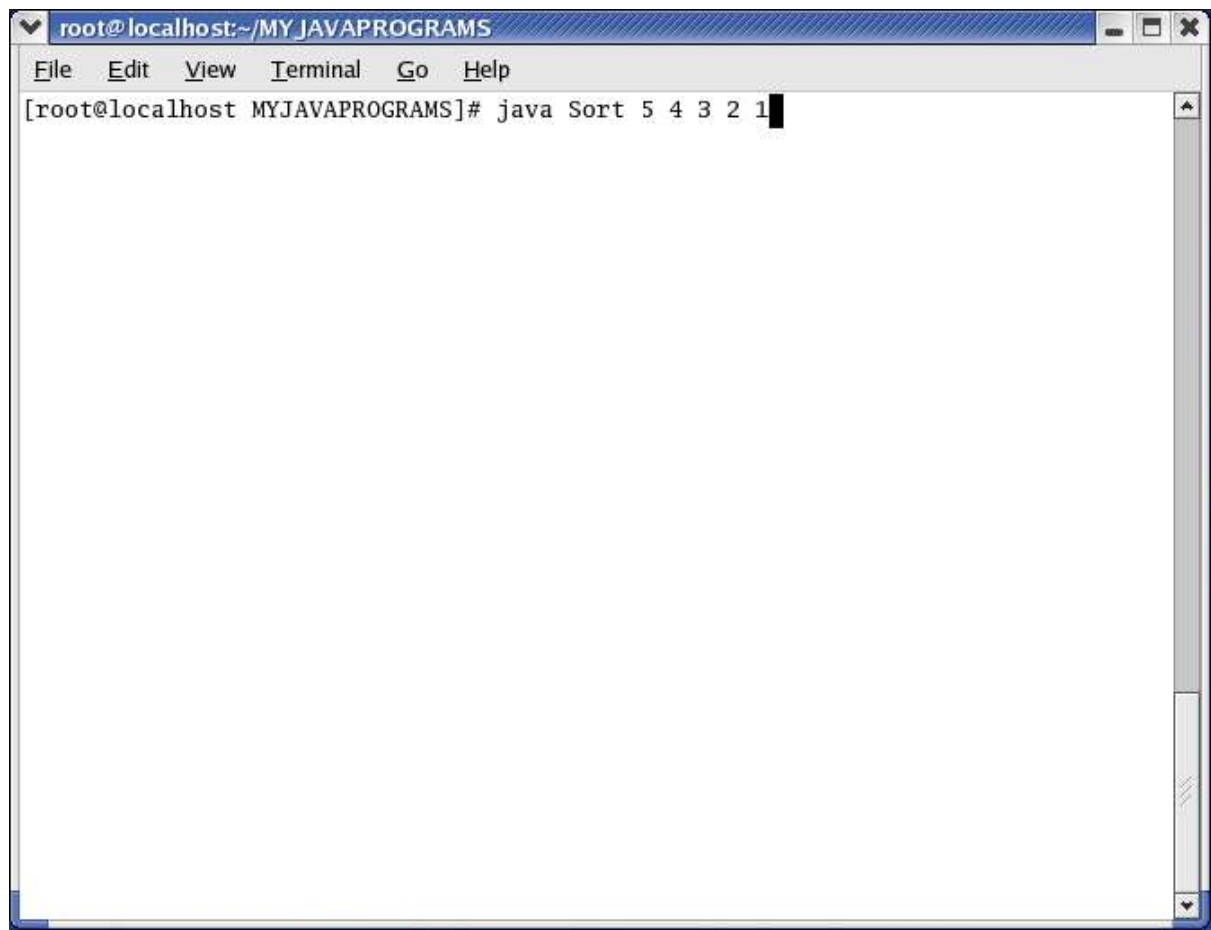
8 Command-line Arguments

In this section, we will study on how to process input from the command-line by using arguments pass onto a Java program.

8.1 *Command-line arguments*

A Java application can accept any number of arguments from the command-line. Command-line arguments allow the user to affect the operation of an application for one invocation. The user enters command-line arguments when invoking the application and specifies them after the name of the class to run.

For example, suppose you have a Java application, called Sort, that sorts five numbers, you run it like this:

A screenshot of a terminal window. The title bar at the top reads 'root@localhost:~/MY_JAVAPROGRAMS'. Below the title bar is a menu bar with 'File', 'Edit', 'View', 'Terminal', 'Go', and 'Help'. The main area of the terminal shows the command prompt '[root@localhost MYJAVAPROGRAMS]# java Sort 5 4 3 2 1' with a cursor at the end of the line. The terminal has a scroll bar on the right side.

```
root@localhost:~/MY_JAVAPROGRAMS
File Edit View Terminal Go Help
[root@localhost MYJAVAPROGRAMS]# java Sort 5 4 3 2 1
```

Take note that the arguments are separated by spaces.

In the Java language, when you invoke an application, the runtime system passes the command-line arguments to the application's main method via an array of Strings. Each String in the array contains one of the command-line arguments. Remember the declaration for the main method,

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

The arguments that are passed to your program are saved into an array of String with the args identifier.

In the previous example, the command-line arguments passed to the Sort application is an array that contains five strings which are: "5", "4", "3", "2" and "1". You can derive the number of command-line arguments with the array's length attribute.

For example,

```
int numberOfArgs = args.length;
```

If your program needs to support a numeric command-line argument, it must convert a String argument that represents a number, such as "34", to a number. Here's a code snippet that converts a command-line argument to an integer,

```
int firstArg = 0;
if (args.length > 0){
    firstArg = Integer.parseInt(args[0]);
}
```

parseInt throws a NumberFormatException (ERROR) if the format of args[0] isn't valid (not a number).

Coding Guidelines:

Before using command-line arguments, always check if the number of arguments before accessing the array elements so that there will be no exception generated.

8.2 Command-line arguments in Netbeans

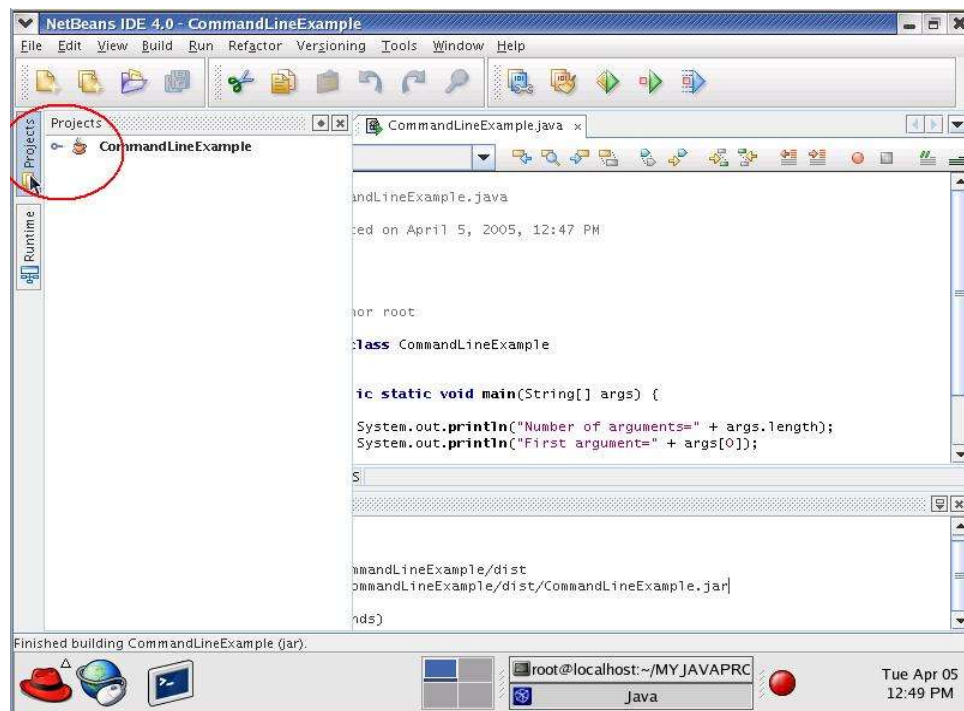
To illustrate on how to pass some arguments to your programs in Netbeans, let us create a Java program that will print the number of arguments and the first argument passed to it.

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

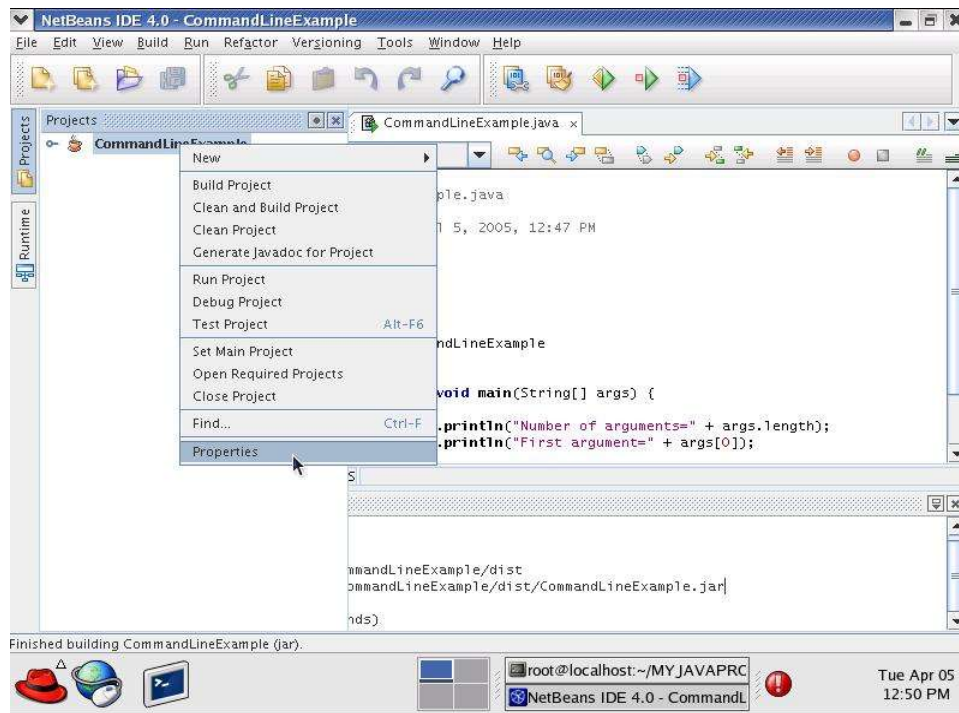
        System.out.println("Number of arguments=" +
                           args.length);
        System.out.println("First Argument="+ args[0]);
    }
}
```

Now, run netbeans and create a new project and name this CommandLineExample. Copy the code shown above and compile the code. Now, follow these steps to pass arguments to your program using Netbeans.

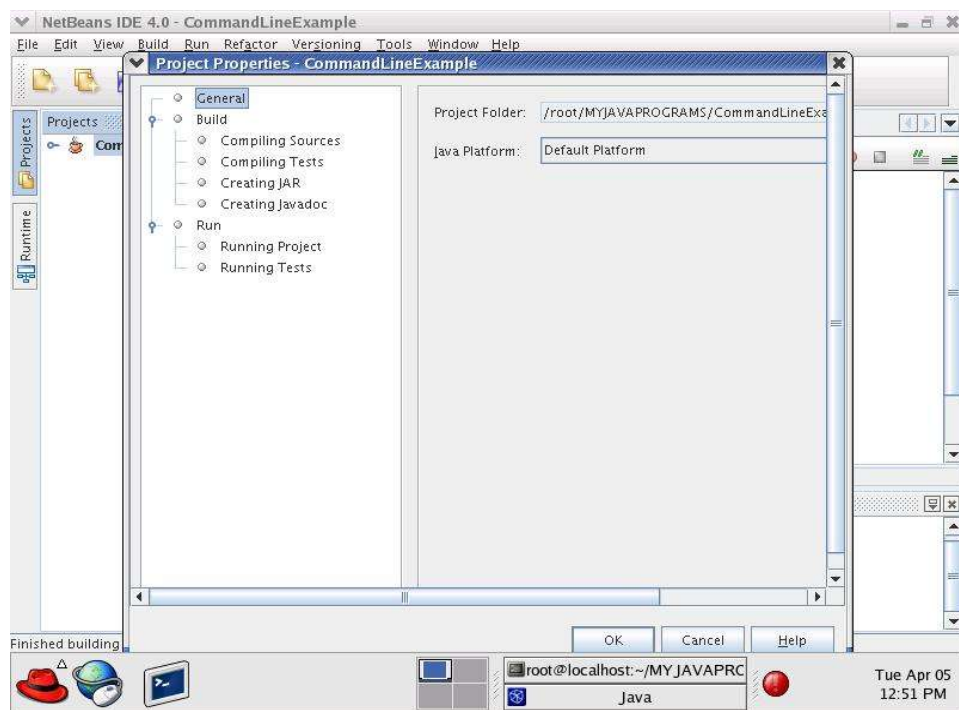
Click on Projects (encircled below).



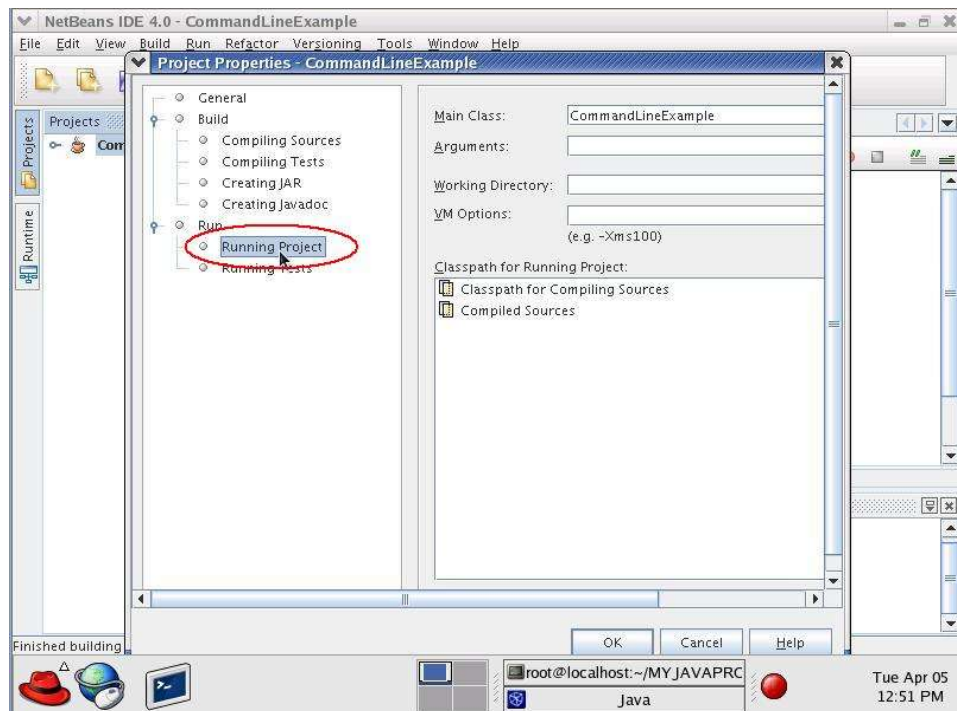
Right-click on the CommandLineExample icon, and a popup menu will appear. Click on Properties.



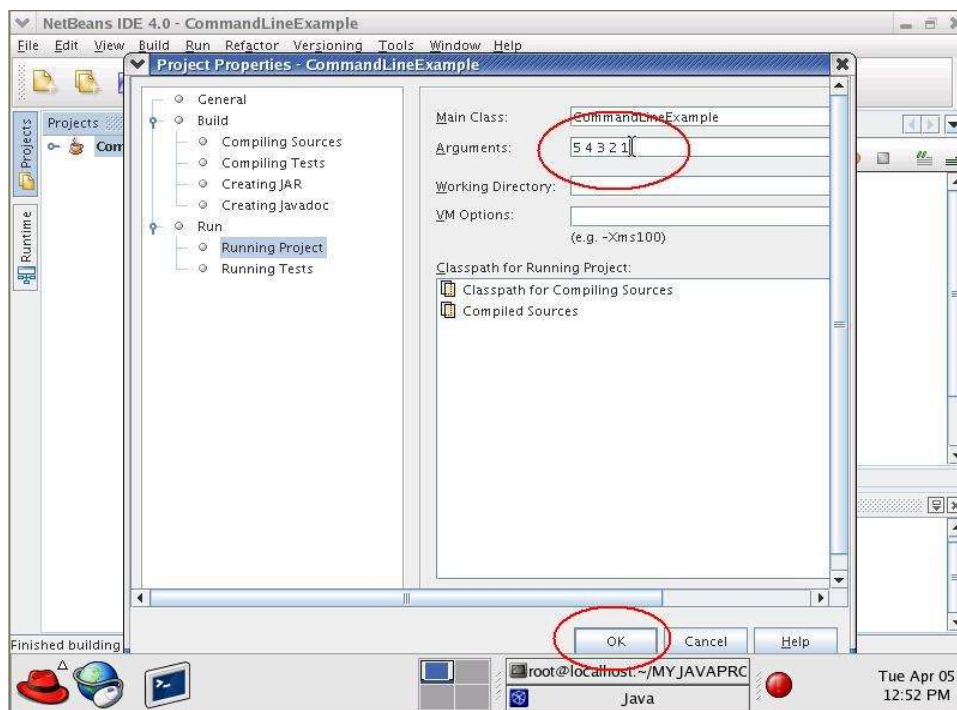
The Project Properties dialog will then appear.



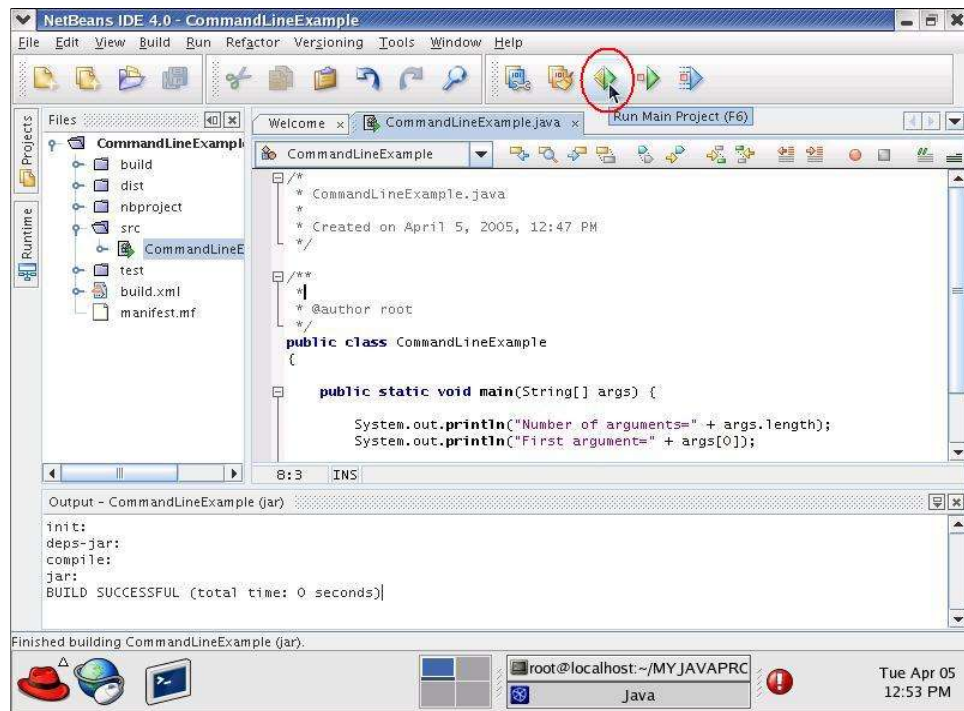
Now, click on Run-> Running Project.



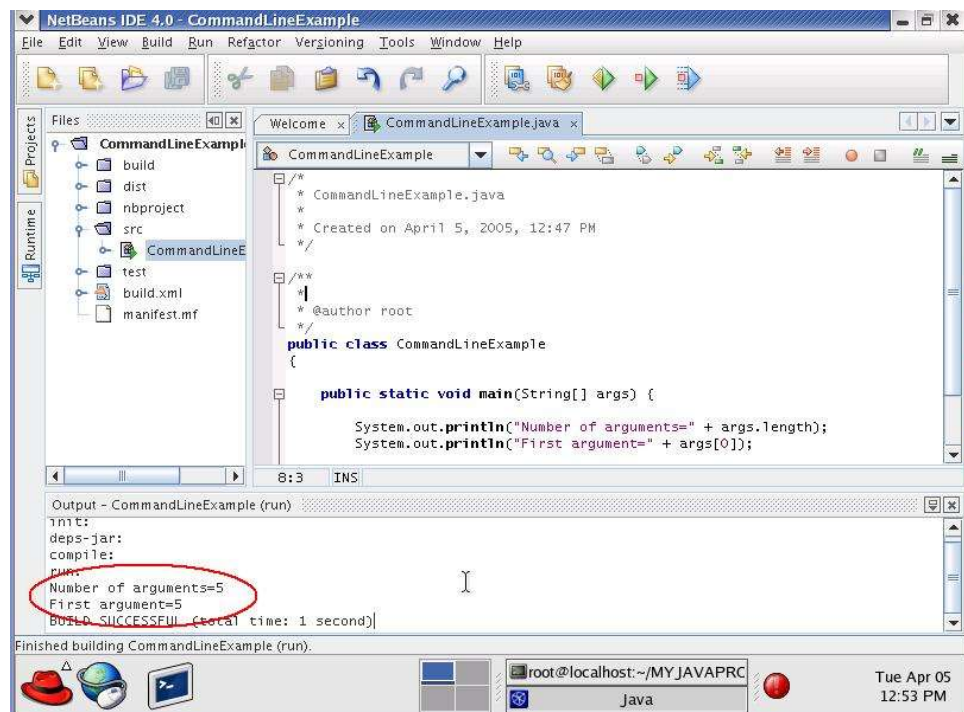
On the Arguments textbox, type the arguments you want to pass to your program. In this case we typed in the arguments 5 4 3 2 1. Then, click on the OK button.



Now try to RUN your program.



As you can see here, the output to your program is the number of arguments which is 5, and the first argument which is 5.



8.3 Exercises

8.3.1 Print arguments

Get input from the user using command-line arguments and print all the arguments to the screen. For example, if the user entered,

```
java Hello world that is all
```

your program should print

```
Hello
world
that
is
all
```

8.3.2 Arithmetic Operations

Get two numbers from the user using command-line arguments and print sum, difference, product and quotient of the two numbers. For example, if the user entered,

```
java ArithmeticOperation 20 4
```

your program should print

```
sum = 24
difference = 16
product = 80
quotient = 5
```

9 Working with the Java Class Library

In this section, we will introduce some basic concepts of object-oriented programming. Later on, we will discuss the concept of classes and objects, and how to use these classes and their members. Comparison, conversion and casting of objects will also be covered. For now, we will focus on using classes that are already defined in the Java class library, we will discuss later on how to create your own classes.

9.1 Introduction to Object-Oriented Programming

Object-Oriented programming or OOP revolves around the concept of **objects** as the basic elements of your programs. When we compare this to the physical world, we can find many objects around us, such as cars, lion, people and so on. These objects are characterized by their **properties** and **behaviors**.

For example, a car object has the properties, *type of transmission*, *manufacturer* and *color*. Its behaviors are *turning*, *braking* and *accelerating*. Similarly, we can define different properties and behavior of a lion. Please refer to the table below for the examples.

Object	Properties	Behavior
Car	type of transmission manufacturer color	turning braking accelerating
Lion	Weight Color hungry or not hungry tamed or wild	roaring sleeping hunting

With these descriptions, the objects in the physical world can easily be modeled as software objects using the **properties as data** and the **behaviors as methods**. These data and methods could even be used in programming games or interactive software to simulate the real-world objects! An example would be a car software object in a racing game or a lion software object in an educational interactive software zoo for kids.

9.2 Classes and Objects

9.2.1 Difference Between Classes and Objects

In the software world, an **object** is a software component whose structure is similar to objects in the real world. Each object is composed of a set of **data** (properties) which are variables describing the essential characteristics of the object, and it also consists of a set of **methods** (behavior) that describes how an object behaves. Thus, an object is a software bundle of variables and related methods. The variables and methods in a Java object are formally known as **instance variables** and **instance methods** to distinguish them from class variables and class methods, which will be discussed later.

The **class** is the fundamental structure in object-oriented programming. It can be thought of as a template, a prototype or a blueprint of an object. It consists of two types of members which are called **fields (properties or attributes)** and methods. Fields specify the data types defined by the class, while methods specify the operations. An object is an **instance** of the class.

To differentiate between classes and objects, let us discuss an example. What we have here is a Car Class which can be used to define several Car Objects. In the table shown below, Car A and Car B are objects of the Car class. The class has *fields* plate number, color, manufacturer, and current speed which are filled-up with corresponding values in objects Car A and Car B. The Car has also some methods Accelerate, Turn and Brake.

Car Class		Object Car A	Object Car B
Instance Variables	Plate Number	ABC 111	XYZ 123
	Color	Blue	Red
	Manufacturer	Mitsubishi	Toyota
	Current Speed	50 km/h	100 km/h
Instance Methods	Accelerate Method		
	Turn Method		
	Brake Method		

When instantiated, each object gets a fresh set of state variables. However, the method implementations are shared among objects of the same class.

Classes provide the benefit of **reusability**. Software programmers can use a class over and over again to create many objects.

9.2.2 Class Variables and Methods

In addition to the instance variables, it is also possible to define **class variables**, which are variables that belong to the whole class. This means that it has the same value for all the objects in the same class. They are also called **static member variables**.

To clearly describe class variables, let's go back to our Car class example. Suppose that our Car class has one class variable called Count. If we change the value of Count to 2, all of the objects of the Car class will have the value 2 for their Count variable.

Car Class		Object Car A	Object Car B
Instance Variables	Plate Number	ABC 111	XYZ 123
	Color	Blue	Red
	Manufacturer	Mitsubishi	Toyota
	Current Speed	50 km/h	100 km/h
Class Variable	Count = 2		
Instance Methods	Accelerate Method		
	Turn Method		
	Brake Method		

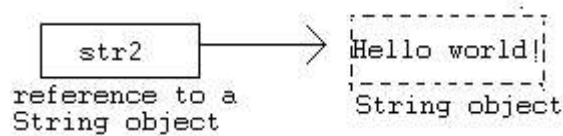
9.2.3 Class Instantiation

To create an object or an instance of a class, we use the **new** operator. For example, if you want to create an instance of the class `String`, we write the following code,

```
String str2 = new String("Hello world!");
```

or also equivalent to,

```
String str2 = "Hello";
```



The `new` operator allocates a memory for that object and returns a **reference** of that memory location to you. When you create an object, you actually invoke the class' **constructor**. The **constructor** is a method where you place all the initializations, it has the same name as the class.

9.3 Methods

9.3.1 What are Methods and Why Use Methods?

In the examples we discussed before, we only have one method, and that is the `main()` method. In Java, we can define many methods which we can call from different methods.

A **method** is a separate piece of code that can be called by a main program or any other method to perform some specific function. The following are characteristics of methods:

- It can return one or no values
- It may accept as many parameters it needs or no parameter at all. Parameters are also called function arguments.
- After the method has finished execution, it goes back to the method that called it.

Now, why do we need to create methods? Why don't we just place all the code inside one big method? The heart of effective problem solving is in problem decomposition. We can do this in Java by creating methods to solve a specific part of the problem. Taking a problem and breaking it into small, manageable pieces is critical to writing large programs.

9.3.2 Calling Instance Methods and Passing Variables

Now, to illustrate how to call methods, let's use the String class as an example. You can use the Java API documentation to see all the available methods in the String class. Later on, we will create our own methods, but for now, let us use what is available.

To call an **instance method**, we write the following,

```
nameOfObject.nameOfMethod( parameters );
```

Let's take two sample methods found in the class String,

Method declaration	Definition
public char charAt(int index)	Returns the character at the specified index. An index ranges from 0 to length() - 1. The first character of the sequence is at index 0, the next at index 1, and so on, as for array indexing.
public boolean equalsIgnoreCase(String anotherString)	Compares this String to another String, ignoring case considerations. Two strings are considered equal ignoring case if they are of the same length, and corresponding characters in the two strings are equal ignoring case.

Using the methods,

```
String    str1 = "Hello";
char      x = str2.charAt(0); //will return the character H
                                //and store it to variable x

String    str2 = "hello";

//this will return a boolean value true
boolean   result = str1.equalsIgnoreCase( str1 );
```

9.3.3 Passing Variables in Methods

In our examples, we already tried passing variables to methods. However, we haven't differentiated between the different types of variable passing in Java. There are two types of passing data to methods, the first one is pass-by-value and then, pass-by-reference.

9.3.3.1 Pass-by-value

When a pass-by-value occurs, the method makes a copy of the value of the variable passed to the method. The method cannot accidentally modify the original argument even if it modifies the parameters during calculations.

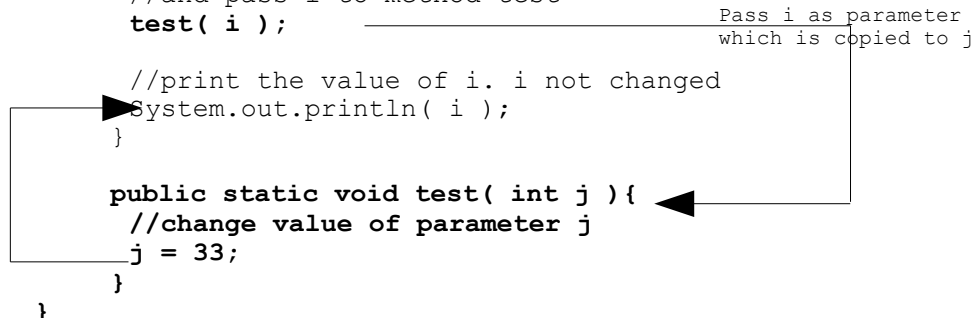
For example,

```
public class TestPassByValue
{
    public static void main( String[] args ){
        int i = 10;
        //print the value of i
        System.out.println( i );

        //call method test
        //and pass i to method test
        test( i );

        //print the value of i. i not changed
        System.out.println( i );
    }

    public static void test( int j ){
        //change value of parameter j
        j = 33;
    }
}
```



In the given example, we called the method test and passed the value of i as parameter. The value of i is copied to the variable of the method j. Since j is the variable changed in the test method, it will not affect the variable value if i in main since it is a different copy of the variable.

By default, all primitive data types when passed to a method are pass-by-value.

9.3.3.2 Pass-by-reference

When a pass-by-reference occurs, the reference to an object is passed to the calling method. This means that, the method makes a copy of the reference of the variable passed to the method. However, unlike in pass-by-value, the method can modify the actual object that the reference is pointing to, since, although different references are used in the methods, the location of the data they are pointing to is the same.

For example,

```
class TestPassByReference
{
    public static void main( String[] args ){
        //create an array of integers
        int []ages = {10, 11, 12};

        //print array values
        for( int i=0; i<ages.length; i++ ){
            System.out.println( ages[i] );
        }

        //call test and pass reference to array
        test( ages );

        //print array values again
        for( int i=0; i<ages.length; i++ ){
            System.out.println( ages[i] );
        }
    }

    public static void test( int[] arr ){
        //change values of array
        for( int i=0; i<arr.length; i++ ){
            arr[i] = i + 50;
        }
    }
}
```

Pass ages as parameter
which is copied to
variable arr

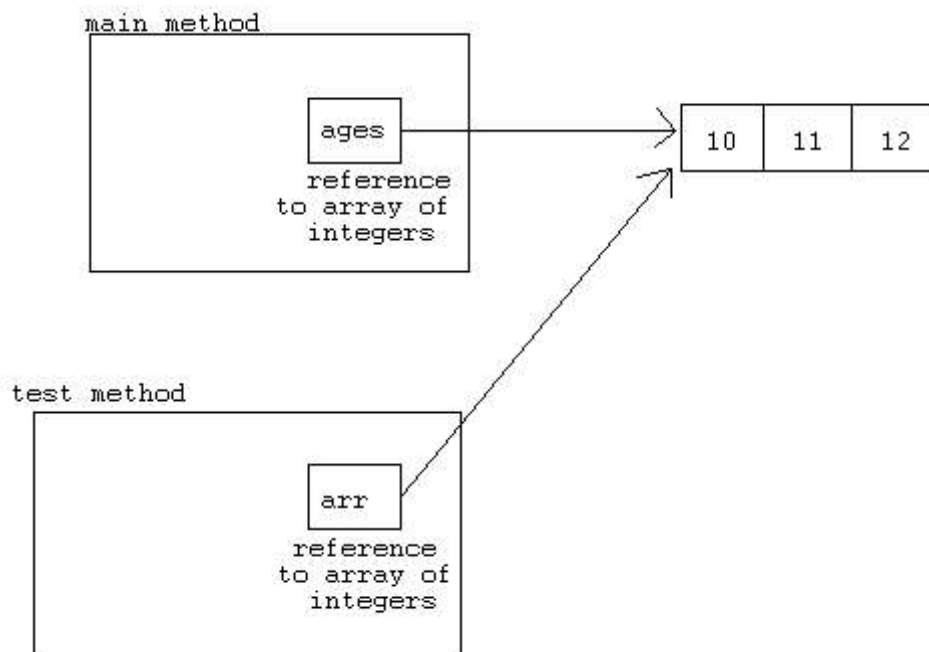


Figure 9.1: Pass-by-reference example

9.3.4 Calling Static Methods

Static methods are methods that can be invoked without instantiating a class (means without invoking the new keyword). Static methods belongs to the class as a whole and not to a certain instance (or object) of a class. Static methods are distinguished from instance methods in a class definition by the keyword `static`.

To call a static method, just type,

```
Classname.staticMethodName(params);
```

Examples of static methods, we've used so far in our examples are,

```
//prints data to screen
System.out.println("Hello world");

//converts the String 10, to an integer
int i = Integer.parseInt("10");

//Returns a String representation of the integer argument as an
//unsigned integer base 16
String hexEquivalent = Integer.toHexString( 10 );
```

9.3.5 Scope of a variable

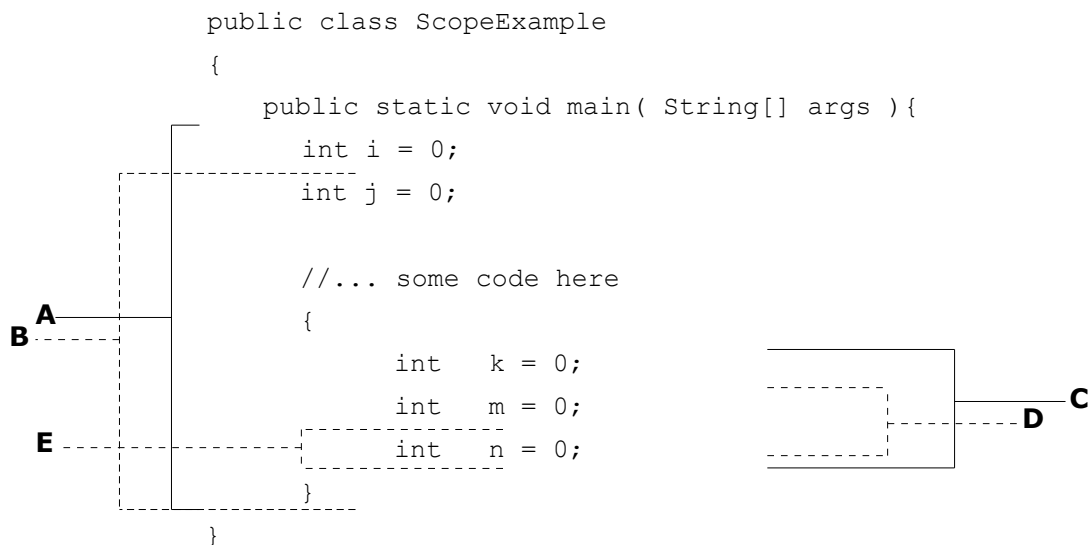
In addition to a variable's data type and name, a variable has scope. The **scope** determines where in the program the variable is accessible. The scope also determines the lifetime of a variable or how long the variable can exist in memory. The scope is determined by where the variable declaration is placed in the program.

To simplify things, just think of the scope as anything between the curly braces {...}. The outer curly braces is called the **outer** blocks, and the inner curly braces is called **inner** blocks.

If you declare variables in the outer block, they are visible (i.e. usable) by the program lines inside the inner blocks. However, if you declare variables in the inner block, you cannot expect the outer block to see it.

A variable's scope is inside the block where it is declared, starting from the point where it is declared, and in the inner blocks.

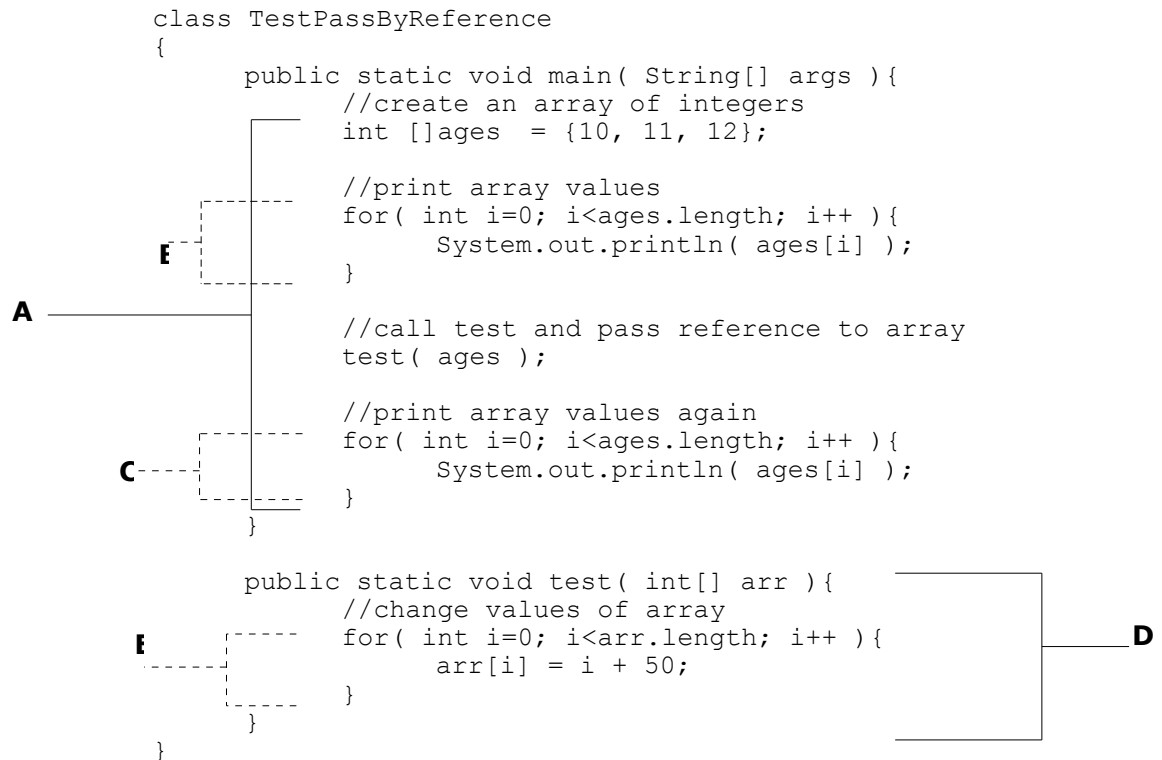
For example, given the following code snippet,



The code we have here represents five scopes indicated by the lines and the letters representing the scope. Given the variables `i`, `j`, `k`, `m` and `n`, and the five scopes `A`, `B`, `C`, `D` and `E`, we have the following scopes for each variable:

- The scope of variable `i` is `A`.
- The scope of variable `j` is `B`.
- The scope of variable `k` is `C`.
- The scope of variable `m` is `D`.
- The scope of variable `n` is `E`.

Now, given the two methods main and test in our previous examples,



In the main method, the scope of the variables are,

ages[] - scope A
i in B - scope B
i in C - scope C

In the test method, the scope of the variables are,

arr[] - scope D
i in E - scope E

When declaring variables, only one variable with a given identifier or name can be declared in a scope. That means that if you have the following declaration,

```
{
    int test = 10;
    int test = 20;
}
```

your compiler will generate an error since you should have unique names for your variables in one block. However, you can have two variables of the same name, if they are not declared in the same block. For example,

```
int test = 0;
System.out.print( test );
//..some code here
{
    int test = 20;
    System.out.print( test );
}
```

When the first `System.out.print` is invoked, it prints the value of the first `test` variable since it is the variable seen at that scope. For the second `System.out.print`, the value 20 is printed since it is the closest `test` variable seen at that scope.

Coding Guidelines:

Avoid having variables of the same name declared inside one method to avoid confusion.

9.4 Casting, Converting and Comparing Objects

In this section, we are going to learn how to do **typecasting**. Typecasting or casting is the process of converting a data of a certain data type to another data type. We will also learn how to convert primitive data types to objects and vice versa. And finally, we are going to learn how to compare objects.

9.4.1 Casting Primitive Types

Casting between primitive types enables you to convert the value of one data from one type to another primitive type. This commonly occurs between numeric types.

There is one primitive data type that we cannot do casting though, and that is the **boolean** data type.

An example of typecasting is when you want to store an integer data to a variable of data type double. For example,

```
int    numInt = 10;
double numDouble = numInt; //implicit cast
```

In this example, since the destination variable (double) holds a larger value than what we will place inside it, the data is implicitly casted to data type double.

Another example is when we want to typecast an int to a char value or vice versa. A character can be used as an int because each character has a corresponding numeric code that represents its position in the character set. If the variable i has the value 65, the cast (char)i produces the character value 'A'. The numeric code associated with a capital A is 65, according to the ASCII character set, and Java adopted this as part of its character support. For example,

```
char valChar = 'A';
int   valInt  = valChar;
System.out.print( valInt ); //explicit cast: output 65
```

When we convert a data that has a large type to a smaller type, we must use an **explicit cast**. Explicit casts take the following form:

```
(dataType) value
```

where,

dataType, is the name of the data type you're converting to

value, is an expression that results in the value of the source type.

For example,

```
double    valDouble = 10.12;
int    valInt = (int)valDouble; //convert valDouble to int type

double    x = 10.2;
int    y = 2;

int    result = (int) (x/y); //typecast result of operation to
int
```

9.4.2 Casting Objects

Instances of classes also can be cast into instances of other classes, with **one restriction: The source and destination classes must be related by inheritance; one class must be a subclass of the other.** We'll cover more about inheritance later.

Analogous to converting a primitive value to a larger type, some objects might not need to be cast explicitly. In particular, because a subclass contains all the same information as its superclass, you can use an instance of a subclass anywhere a superclass is expected.

For example, consider a method that takes two arguments, one of type `Object` and another of type `Window`. You can pass an instance of any class for the `Object` argument because all Java classes are subclasses of `Object`. For the `Window` argument, you can pass in its subclasses, such as `Dialog`, `FileDialog`, and `Frame`. This is true anywhere in a program, not just inside method calls. If you had a variable defined as class `Window`, you could assign objects of that class or any of its subclasses to that variable without casting.

This is true in the reverse, and you can use a superclass when a subclass is expected. There is a catch, however: Because subclasses contain more behavior than their superclasses, there's a loss in precision involved. Those superclass objects might not have all the behavior needed to act in place of a subclass object. For example, if you have an operation that calls methods in objects of the class `Integer`, using an object of class `Number` won't include many methods specified in `Integer`. Errors occur if you try to call methods that the destination object doesn't have.

To use superclass objects where subclass objects are expected, you must cast them explicitly. You won't lose any information in the cast, but you gain all the methods and variables that the subclass defines. To cast an object to another class, you use the same operation as for primitive types:

To cast,

```
(classname) object
```

where,

classname, is the name of the destination class
object, is a reference to the source object.

- **Note:** that casting creates a reference to the old object of the type *classname*; the old object continues to exist as it did before.

The following example casts an instance of the class `VicePresident` to an instance of the class `Employee`; `VicePresident` is a subclass of `Employee` with more information, which here defines that the `VicePresident` has executive washroom privileges,

```
Employee emp = new Employee();
VicePresident veep = new VicePresident();
emp = veep; // no cast needed for upward use
veep = (VicePresident)emp; // must cast explicitlyCasting
```

9.4.3 Converting Primitive Types to Objects and Vice Versa

One thing you can't do under any circumstance is cast from an object to a primitive data type, or vice versa. Primitive types and objects are very different things in Java, and you can't automatically cast between the two or use them interchangeably.

As an alternative, the **java.lang** package includes classes that correspond to each primitive data type: Float, Boolean, Byte, and so on. Most of these classes have the same names as the data types, except that the class names begin with a capital letter (Short instead of short, Double instead of double, and the like). Also, two classes have names that differ from the corresponding data type: Character is used for char variables and Integer for int variables. **(Called Wrapper Classes)**

Java treats the data types and their class versions very differently, and a program won't compile successfully if you use one when the other is expected.

Using the classes that correspond to each primitive type, you can create an object that holds the same value.

Examples:

```
//The following statement creates an instance of the Integer
// class with the integer value 7801 (primitive -> Object)
Integer dataCount = new Integer(7801);

//The following statement converts an Integer object to
// its primitive data type int. The result is an int with
//value 7801
int newCount = dataCount.intValue();

// A common translation you need in programs
// is converting a String to a numeric type, such as an int
// Object->primitive
String pennsylvania = "65000";
int penn = Integer.parseInt(pennsylvania);
```

- **CAUTION:** The Void class represents nothing in Java, so there's no reason it would be used when translating between primitive values and objects. It's a placeholder for the void keyword, which is used in method definitions to indicate that the method does not return a value.

9.4.4 Comparing Objects

In our previous discussions, we learned about operators for comparing values—equal, not equal, less than, and so on. Most of these operators work only on primitive types, not on objects. If you try to use other values as operands, the Java compiler produces errors.

The exceptions to this rule are the operators for equality: `==` (equal) and `!=` (not equal). When applied to objects, these operators don't do what you might first expect. Instead of checking whether one object has the same value as the other object, they determine whether both sides of the operator refer to the same object.

To compare instances of a class and have meaningful results, you must implement special methods in your class and call those methods. A good example of this is the `String` class.

It is possible to have two different `String` objects that contain the same values. If you were to employ the `==` operator to compare these objects, however, they would be considered unequal. Although their contents match, they are not the same object.

To see whether two `String` objects have matching values, a method of the class called `equals()` is used. The method tests each character in the string and returns `true` if the two strings have the same values.

The following code illustrates this,

```
class EqualsTest {
    public static void main(String[] arguments) {
        String str1, str2;
        str1 = "Free the bound periodicals.";
        str2 = str1;

        System.out.println("String1: " + str1);
        System.out.println("String2: " + str2);
        System.out.println("Same object? " + (str1 == str2));

        str2 = new String(str1);

        System.out.println("String1: " + str1);
        System.out.println("String2: " + str2);
        System.out.println("Same object? " + (str1 == str2));
        System.out.println("Same value? " + str1.equals(str2));
    }
}
```

This program's output is as follows,

OUTPUT:

```
String1: Free the bound periodicals.
String2: Free the bound periodicals.
Same object? true
String1: Free the bound periodicals.
String2: Free the bound periodicals.
Same object? false
Same value? True
```

Now let's discuss the code.

```
String str1, str2;  
str1 = "Free the bound periodicals.";
```

The first part of this program declares two variables (`str1` and `str2`), assigns the literal "Free the bound periodicals." to `str1`, and then assigns that value to `str2`. As you learned earlier, `str1` and `str2` now point to the same object, and the equality test proves that.

```
str2 = new String(str1);
```

In the second part of this program, you create a new `String` object with the same value as `str1` and assign `str2` to that new `String` object. Now you have two different string objects in `str1` and `str2`, both with the same value. Testing them to see whether they're the same object by using the `==` operator returns the expected answer: `false`—they are not the same object in memory. Testing them using the `equals()` method also returns the expected answer: `true`—they have the same values.

- **NOTE:** Why can't you just use another literal when you change `str2`, rather than using `new`? String literals are optimized in Java; if you create a string using a literal and then use another literal with the same characters, Java knows enough to give you the first `String` object back. Both strings are the same objects; you have to go out of your way to create two separate objects.

9.4.5 Determining the Class of an Object

Want to find out what an object's class is? Here's the way to do it for an object assigned to the variable `key`:

1. The **`getClass()` method** returns a `Class` object (where `Class` is itself a class) that has a method called `getName()`. In turn, `getName()` returns a string representing the name of the class.

For Example,

```
String name = key.getClass().getName();
```

2. The `instanceOf` operator

The `instanceOf` has two operands: a reference to an object on the left and a class name on the right. The expression returns `true` or `false` based on whether the object is an instance of the named class or any of that class's subclasses.

For Example,

```
boolean ex1 = "Texas" instanceof String; // true  
Object pt = new Point(10, 10);  
boolean ex2 = pt instanceof String; // false
```

9.5 Exercises

9.5.1 Defining terms

In your own words, define the following terms:

1. Class
2. Object
3. Instantiate
4. Instance Variable
5. Instance Method
6. Class Variables or static member variables
7. Constructor

9.5.2 Java Scavenger Hunt

Pipoy is a newbie in the Java programming language. He just heard that there are already ready-to-use APIs in Java that one could use in their programs, and he's eager to try them out. The problem is, Pipoy does not have a copy of the Java Documentation, and he also doesn't have an internet access, so there's no way for him to view the Java APIs.

Your task is to help Pipoy look for the APIs (Application Programming Interface). You should state the class where the method belongs, the method declaration and a sample usage of the said method.

For example, if Pipoy wants to know the method that converts a String to integer, your answer should be:

Class: Integer

Method Declaration: public static int parseInt(String value)

Sample Usage:

```
String    strValue = "100";  
int      value = Integer.parseInt( strValue );
```

Make sure that the snippet of code you write in your sample usage compiles and outputs the correct answer, so as not to confuse Pipoy. **(Hint: All methods are in the java.lang package).** In cases where you can find more methods that can accomplish the task, give only one.

Now let's start the search!

1. Look for a method that checks if a certain String ends with a certain suffix. For example, if the given string is "Hello", the method should return true the suffix given is "lo", and false if the given suffix is "alp".
2. Look for the method that determines the character representation for a specific digit in the specified radix. For example, if the input digit is 15, and the radix is 16, the method would return the character F, since F is the hexadecimal representation for the number 15 (base 10).
3. Look for the method that terminates the currently running Java Virtual Machine
4. Look for the method that gets the floor of a double value. For example, if I input a 3.13, the method should return the value 3.
5. Look for the method that determines if a certain character is a digit. For example, if I input '3', it returns the value true.

10 Creating your own Classes

Now that we've studied on how to use existing classes from the Java class library, we will now be studying on how to write our own classes. For this section, in order to easily understand how to create classes, we will make a sample class wherein we will add more data and functionality as we go along the way.

We will create a class that contains information of a Student and operations needed for a certain student record.

10.1 Defining your own classes

Before writing your class, think first on where you will be using your class and how your class will be used. Think of an appropriate name for the class, and list all the information or properties that you want your class to contain. Also list down the methods that you will be using for your class.

In this section, we will be creating a class that will contain a student record. Since we've already identified the purpose of our class, we can now name it. An appropriate name for our class would be `StudentRecord`.

Now, to define our class we write,

```
public class StudentRecord
{
    //we'll add more code here later
}
```

where,

- `public` - means that our class is accessible to other classes outside the package
- `class` - this is the keyword used to create a class in Java
- `StudentRecord` - a unique identifier that describes our class

Coding Guidelines:

1. Think of an appropriate name for your class. Don't just call your class XYZ or any random names you can think of.
2. Class names should start with a CAPITAL letter.
3. The filename of your class should have the SAME NAME as your class name.

10.2 Declaring Fields

Now, that we have defined our class, let us now write down a list of information that a student record can contain. For each information, also list what data types would be appropriate to use. For example, you don't want to have a data type `int` for a student's name, or a `String` for a student's grade. The following are some sample information we want to add to the student record.

name	- String
address	- String
age	- int
math grade	- double
english grade	- double
science grade	- double
average grade	- double

You can add more information if you want to, it's all really up to you. But for this example, we will be using these information.

10.2.1 Instance Variables

Now that we have a list of all the properties we want to add to our class, let us now add them to our code. Since we want these properties to be unique for each object (or for each student), we should declare them as instance variables. For example,

```
public class StudentRecord
{
    private String    name;
    private String    address;
    private int       age;
    private double    mathGrade;
    private double    englishGrade;
    private double    scienceGrade;
    private double    average;

    //we'll add more code here later
}
```

where,

`private` here means that the variables are only accessible within the class. Other objects cannot access these variables directly. We will cover more about accessibility later.

Coding Guidelines:

1. Declare all your instance variables on the top of the class declaration.
2. Declare one variable for each line.
3. Instance variables, like any other variables should start with a SMALL letter.
4. Use an appropriate data type for each variable you declare.
5. Declare instance variables as `private` so that only class methods can access them directly.

10.2.2 Class Variables or Static Variables

Aside from instance variables, we can also declare class variables or variables that belong to the class as a whole. The value of these variables are the same for all the objects of the same class. Now suppose, we want to know the total number of student records we have for the whole class, we can declare one static variable that will hold this value. Let us call this as studentCount.

To declare a static variable,

```
public class StudentRecord
{
    //instance variables we have declared

    private static int studentCount;

    //we'll add more code here later
}
```

we use the keyword static to indicate that a variable is a static variable.

So far, our whole code now looks like this.

```
public class StudentRecord
{
    private String    name;
    private String    address;
    private int       age;
    private double    mathGrade;
    private double    englishGrade;
    private double    scienceGrade;
    private double    average;

    private static int studentCount;

    //we'll add more code here later
}
```

10.3 Declaring Methods

Now that we have our class, we'll add now more functionality that we will be needing for the class. We can do this by creating methods for the class. Let's first list down the set of functionalities that we want our class to have.

We want to access information about the student like what the student's name is, what's his/her address and other information. However, we declared these variables to be private, so they are only accessible within the class. What we can do is create what we call **accessor methods** which are used to read values from our class variables. We also create **mutator methods** which are used to write or change values of our class variables. You can restrict access to a class variable by not creating an accessor/mutator method for it.

For our example, we want an accessor and mutator method that can read and change the name, address, english grade, math grade and science grade of the student. We also want to create a method that can compute the average grade and return the result. For the student count, we want a method that can read the value of the student count but cannot modify it. To summarize, we have the list of accessor and mutator methods and what it can do,

name	- read, write
address	- read, write
english grade	- read, write
math grade	- read, write
science grade	- read, write
average grade	- compute/read
studentCount	- read

An accessor method usually starts with a **get[NameOfInstanceVariable]** if we want to read the value contained in an instance variable. A mutator method that changes the value of the instance variable, we write it as **set[NameOfInstanceVariable]**. So, given the list above, we can have the following names for our accessor and mutator methods,

name	- getName, setName
address	- getAddress, setAddress
english grade	- getEnglishGrade, setEnglishGrade
math grade	- getMathGrade, setMathGrade
science grade	- getScienceGrade, setScienceGrade
average grade	- getAverageGrade
studentCount	- getStudentCount

10.3.1.1 get() Methods

Now let's take a look at one implementation of a get() method,

```
public class StudentRecord
{
    private String    name;
    :
    :
    public String getName() {
        return name;
    }
}
```

where,

- | | |
|---------|---|
| public | - means that the method can be called from objects outside the class |
| String | - is the return type of the method. This means that the method should return a value of type String |
| getName | - the name of the method |
| () | - this means that our method does not have any parameters |

The statement,

```
return name;
```

in our program signifies that it will return the value of the instance variable name to the calling method. Take note that the return type of the method should have the same data type as the data in the return statement. You usually encounter the following error if the two does not have the same data type,

```
StudentRecord.java:14: incompatible types
found   : int
required: java.lang.String
        return age;
        ^
1 error
```

10.3.1.2 Multiple return statements

You can have multiple return statements for a method as long as they are not on the same block. You can also use constants to return values instead of variables. For example, consider the method,

```
public String getNumberInWords( int num ){  
    String defaultNum = "zero";  
    if( num == 1 ){  
        return "one"; //return a constant  
    }  
    else if( num == 2){  
        return "two"; //return a constant  
    }  
  
    //return a variable  
    return defaultNum;  
}
```

10.3.1.3 set() Methods

Now let's take a look at one implementation of a set() method,

```
public class StudentRecord  
{  
    private String    name;  
    :  
    :  
    public void setName( String temp ){  
        name = temp;  
    }  
}
```

where,

public	- means that the method can be called from objects outside the class
void	- means that the method does not return any value
setName	- the name of the method
(String temp)	- parameter that will be used inside our method

The statement,

```
name = temp;
```

assigns the value of temp to name and thus changes the data inside the instance variable name.

Take note that set methods don't return values. However, it contains some program argument or arguments that will be used inside the method.

From the given examples, you can now create the other accessor and mutator methods for the other instance variables. But before we show the whole code, let us first take a look at the `getAverage` method.

```
public class StudentRecord
{
    private String      name;
    :
    :
    public double getAverage(){

        double result = 0;
        result = ( mathGrade+englishGrade+scienceGrade )/3;

        return result;
    }
}
```

The `getAverage` method computes the average of the 3 grades and returns the result.

Coding Guidelines:

1. Method names should start with a *SMALL* letter.
2. Method names should be verbs
3. Always provide documentation before the declaration of the method. You can use *javadocs* style for this. Please see example.

For the static variable `studentCount`, we can create a static method to access its value.

```
public class StudentRecord
{
    :
    private static int studentCount;
    :
    :
    public static int getStudentCount(){
        return studentCount;
    }
}
```

where,

<code>public</code>	- means that the method can be called from objects outside the class
<code>static</code>	- means that the method is static and should be called by typing, <code>[ClassName].[methodName]</code> . For example, in this case, we call the method <code>StudentRecord.getStudentCount()</code>
<code>int</code>	- is the return type of the method. This means that the method should return a value of type <code>int</code>
<code>getStudentCount</code>	- the name of the method
<code>()</code>	- this means that our method does not have any parameters

For now, `getStudentCount` will always return the value zero since we haven't done anything yet in our program in order to set its value. We will try to change the value of `studentCount` later on when we discuss constructors.

Here is the code for our StudentRecord class:

```
public class StudentRecord
{
    private String    name;
    private String    address;
    private int       age;
    private double    mathGrade;
    private double    englishGrade;
    private double    scienceGrade;
    private double    average;

    private static int studentCount;

    /**
     * Returns the name of the student
     */
    public String getName(){
        return name;
    }

    /**
     * Changes the name of the student
     */
    public void setName( String temp ){
        name = temp;
    }

    // other code here ....

    /**
     * Computes the average of the english, math and science
     * grades
     */
    public double getAverage(){

        double result = 0;
        result = ( mathGrade+englishGrade+scienceGrade )/3;

        return result;
    }

    /**
     * returns the number of instances of StudentRecords
     */
    public static int getStudentCount(){
        return studentCount;
    }
}
```

Now, here's a sample code of a class that uses our StudentRecord class.

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

        //create three objects for Student record
        StudentRecord    annaRecord = new StudentRecord();
        StudentRecord    beahRecord = new StudentRecord();
        StudentRecord    crisRecord = new StudentRecord();

        //set the name of the students
        annaRecord.setName("Anna");
        beahRecord.setName("Beah");
        crisRecord.setName("Cris");

        //print anna's name
        System.out.println( annaRecord.getName() );

        //print number of students
        System.out.println("Count="+StudentRecord.getStudentCount
    ());
    }
}
```

The output of this program is,

```
Anna
Student Count = 0
```

10.4 The *this* reference

The **this** reference is used to access the instance variables shadowed by the parameters. To understand this better, let's take for example the `setAge` method. Suppose we have the following declaration for `setAge`.

```
public void setAge( int age ){
    age = age; //WRONG!!!
}
```

The parameter name in this declaration is `age`, which has the same name as the instance variable `age`. Since the parameter `age` is the closest declaration to the method, the value of the parameter `age` will be used. So in the statement,

```
age = age;
```

we are just assigning the value of the parameter `age` to itself! This is not what we want to happen in our code. In order to correct this mistake, we use the **this** reference. To use the `this` reference, we type,

```
this.[nameOfTheInstanceVariable]
```

So for example, we can now rewrite our code to,

```
public void setAge( int age ){
    this.age = age;
}
```

This method will then assign the value of the parameter `age` to the instance variable of the object `StudentRecord`.

Take note that, you can only use the `this` reference for instance variables and NOT static or class variables.

10.5 Overloading Methods

In our classes, we want to sometimes create methods that has the same names but function differently depending on the parameters that are passed to them. This capability is possible in Java, and it is called **Method Overloading**.

Method overloading allows a method with the same name but different parameters, to have different implementations and return values of different types. Rather than invent new names all the time, method overloading can be used when the same operation has different implementations.

For example, in our StudentRecord class we want to have a method that prints information about the student. However, we want the print method to print things differently depending on the parameters we pass to it. For example, when we pass a String, we want the print method to print out the name, address and age of the student. When we pass 3 double values, we want the method to print the student's name and grades.

We have the following overloaded methods inside our StudentRecord class,

```
public void print( String temp )
{
    System.out.println("Name:" + name);
    System.out.println("Address:" + address);
    System.out.println("Age:" + age);
}

public void print(double eGrade, double mGrade, double sGrade)
{
    System.out.println("Name:" + name);
    System.out.println("Math Grade:" + mGrade);
    System.out.println("English Grade:" + eGrade);
    System.out.println("Science Grade:" + sGrade);
}
```

When we try to call this in the following main method,

```
public static void main( String[] args )
{
    StudentRecord    annaRecord = new StudentRecord();

    annaRecord.setName("Anna");
    annaRecord.setAddress("Philippines");
    annaRecord.setAge(15);
    annaRecord.setMathGrade(80);
    annaRecord.setEnglishGrade(95.5);
    annaRecord.setScienceGrade(100);

    //overloaded methods
    annaRecord.print( annaRecord.getName() );
    annaRecord.print( annaRecord.getEnglishGrade(),
                      annaRecord.getMathGrade(),
                      annaRecord.getScienceGrade() );
}
```

we will have the output for the first call to print,

```
Name:Anna
Address:Philippines
Age:15
```

we will have the output for the second call to print,

```
Name:Anna
Math Grade:80.0
English Grade:95.5
Science Grade:100.0
```

Always remember that overloaded methods have the following properties,

- the same name
- **different parameters**
- return types can be different or the same

10.6 Declaring Constructors

We have discussed before the concept of constructors. Constructors are important in instantiating an object. It is a method where all the initializations are placed.

The following are the properties of a constructor:

1. Constructors have the **same name as the class**
2. A constructor is just like an ordinary method, however only the following information can be placed in the header of the constructor, scope or accessibility identifier (like public...), constructor's name and parameters if it has any.
3. Constructors **does not have any return value**
4. **You cannot call a constructor directly**, it can only be called by using the **new** operator during class instantiation.

Every class has a default constructor. The **default constructor** is the constructor without any parameters. If the class does not specify any constructors, then an implicit default constructor is created.

For example, in our StudentRecord class, the default constructor would look like this,

```
public StudentRecord()
{
    //some code here
}
```

As we have mentioned, constructors can also be overloaded, for example, we have here four overloaded constructors,

```
public StudentRecord(){
    //some initialization code here
}

public StudentRecord(String temp){
    this.name = temp;
}

public StudentRecord(String name, String address){
    this.name = name;
    this.address = address;
}

public StudentRecord(double mGrade, double eGrade,
    double sGrade){
    mathGrade = mGrade;
    englishGrade = eGrade;
    scienceGrade = sGrade;
}
```

To use these constructors, we have the following code,

```
public static void main( String[] args )
{
    //create three objects for Student record
    StudentRecord    annaRecord=new StudentRecord("Anna");

    StudentRecord    beahRecord=new StudentRecord
("Beah","Philippines");
    StudentRecord    crisRecord=new StudentRecord
(80,90,100);

    //some code here
}
```

Now, before we move on, let us go back to the static variable `studentCount` we have declared a while ago. The purpose of the `studentCount` is to count the number of objects that are instantiated with the class `StudentRecord`. So, what we want to do here is, everytime an object of class `StudentRecord` is instantiated, we increment the value of `studentCount`. A good location to modify and increment the value of `studentCount` is in the constructors, because it is always called everytime an object is instantiated. For example,

```
public StudentRecord(){
    //some initialization code here
    studentCount++; //add a student
}

public StudentRecord(String temp){
    this.name = temp;
    studentCount++; //add a student
}

public StudentRecord(String name, String address){
    this.name = name;
    this.address = address;
    studentCount++; //add a student
}

public StudentRecord(double mGrade, double eGrade,
                      double sGrade){
    mathGrade = mGrade;
    englishGrade = eGrade;
    scienceGrade = sGrade;
    studentCount++; //add a student
}
```


10.7 The *this()* Constructor Call

Constructor calls can be chained, meaning, you can call another constructor from inside another constructor. We use the **this()** call for this. For example, given the following code,

```
1: public StudentRecord(){
2:     this("some string");
3:
4: }
5:
6: public StudentRecord(String temp){
7:     this.name = temp;
8: }
9:
10: public static void main( String[] args )
11: {
12:
13:     StudentRecord    annaRecord = new StudentRecord();
14: }
```

Given the code above, when the statement at line 13 is called, it will call the default constructor line 1. When statement in line 2 is executed, it will then call the constructor that has a String parameter (in line 6).

There are a few things to remember when using the **this** constructor call:

1. When using the this constructor call, **IT MUST OCCUR AS THE FIRST STATEMENT in a constructor**
2. It can **ONLY BE USED IN A CONSTRUCTOR DEFINITION**. The this call can then be followed by any other relevant statements.

10.8 Packages

Packages are Java's means of grouping related classes and interfaces together in a single unit (interfaces will be discussed later). This powerful feature provides for a convenient mechanism for managing a large group of classes and interfaces while avoiding potential naming conflicts.

10.8.1 Importing Packages

To be able to use classes outside of the package you are currently working in, you need to import the package of those classes. By default, all your Java programs import the `java.lang.*` package, that is why you can use classes like `String` and `Integers` inside the program even though you haven't imported any packages.

The syntax for importing packages is as follows,

```
import nameOfPackage;
```

For example, if you want to use the class `Color` inside package `awt`, you have to type the following,

```
import java.awt.Color;
import java.awt.*;
```

The first statement imports the specific class `Color` while the other imports all of the classes in the `java.awt` package.

Another way to import classes from other packages is through explicit package referencing. This is done by using the package name to declare an object of a class.

```
java.awt.Color color;
```

10.8.2 *Creating your own packages*

Suppose we want to create a package where we will place our StudentRecord class, together with other related classes. We will call our package, SchoolClasses.

The first thing you have to do is create a folder named SchoolClasses. Copy all the classes that you want to belong to this package inside this folder. After copying, add the following code at the top of the class file. For example,

```
package SchoolClasses;

public class StudentRecord
{
    private String    name;
    private String    address;
    private int       age;
    :
    :
}
```

Packages can also be nested. In this case, the Java interpreter expects the directory structure containing the executable classes to match the package hierarchy.

10.9 Access Modifiers

When creating our classes and defining the properties and methods in our class, we want to implement some kind of restriction to access these data. For example, if you want a certain attribute to be changed only by the methods inside the class, you may want to hide this from other objects using your class. In Java, we have what we call **access modifiers** in order to implement this.

There are four different types of member access modifiers in Java: public, private, protected and default. The first three access modifiers are explicitly written in the code to indicate the access type, for the fourth one which is default, no keyword is used.

10.9.1 default access (also called package accessibility)

This specifies that only classes in the same package can have access to the class' variables and methods. There are no actual keyword for the default modifier; it is applied in the absence of an access modifier. For example,

```
public class StudentRecord
{
    //default access to instance variable
    int    name;

    //default access to method
    String getName() {
        return name;
    }
}
```

In this example, the instance variable name and the method getName() can be accessed from other objects, as long as the object belongs to the same package where the class StudentRecord belongs to.

10.9.2 public access

This specifies that class members are accessible to anyone, both inside and outside the class. Any object that interacts with the class can have access to the public members of the class. For example,

```
public class StudentRecord
{
    //default access to instance variable
    public int name;

    //default access to method
    public String getName(){
        return name;
    }
}
```

In this example, the instance variable `name` and the method `getName()` can be accessed from other objects.

10.9.3 *protected access*

This specifies that the class members are accessible only to methods in that class and the subclasses of the class. For example,

```
public class StudentRecord
{
    //default access to instance variable
    protected int    name;

    //default access to method
    protected String getName() {
        return name;
    }
}
```

In this example, the instance variable `name` and the method `getName()` can be accessed only from methods inside the class and from subclasses of `StudentRecord`. We will discuss about subclasses on the next chapter.

10.9.4 private access

This specifies that the class members are only accessible by the class they are defined in. For example,

```
public class StudentRecord
{
    //default access to instance variable
    private int name;

    //default access to method
    private String getName() {
        return name;
    }
}
```

In this example, the instance variable name and the method getName() can be accessed only from methods inside the class.

Coding Guidelines:

The instance variables of a class should normally be declared private, and the class will just provide accessor and mutator methods to these variables.

10.10 Exercises

10.10.1 Address Book Entry

Your task is to create a class that contains an address book entry. The following table describes the information that an addressbook entry has.

Attributes/Properties	Description
Name	Name of the person in the addressbook
Address	Address of the person
Telephone Number	Telephone number of the person
Email Address	Person's Email address

For the methods, create the following:

1. Provide the necessary accessor and mutator methods for all the attributes.
2. Constructors

10.10.2 AddressBook

Create a class address book that can contain 100 entries of AddressBookEntry objects (use the class you created in the first exercise). You should provide the following methods for the address book.

1. Add entry
2. Delete entry
3. View all entries
4. Update an entry

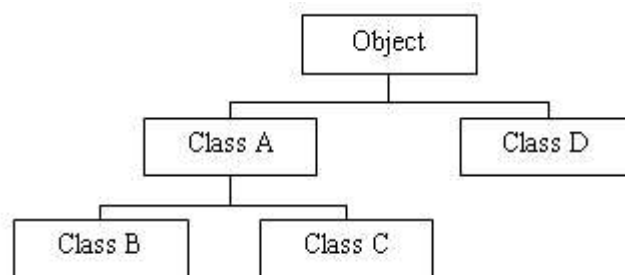
11 Inheritance, Polymorphism and Interfaces

In this section, we will be discussing on how a class can inherit the properties of an existing class. A class that does this is called a subclass and its parent class is called the superclass. We will also be discussing a special property of Java wherein it can automatically apply the proper methods to each object regardless of what subclass the object came from. This property is known as polymorphism. Finally, we are going to discuss about interfaces that helps reduce programming effort.

11.1 Inheritance

In Java, all classes, including the classes that make up the Java API, are subclassed from the **Object** superclass. A sample class hierarchy is shown below.

Any class above a specific class in the class hierarchy is known as a **superclass**. While any class below a specific class in the class hierarchy is known as a subclass of that class.



Class hierarchy in Java.

Inheritance is a major advantage in object-oriented programming since once a behavior (method) is defined in a superclass, that behavior is automatically inherited by all subclasses. Thus, you can encode a method only once and they can be used by all subclasses. A subclass only need to implement the differences between itself and the parent.

11.1.1 Defining Superclasses and Subclasses

To derive a class, we use the **extends** keyword. In order to illustrate this, let's create a sample parent class. Suppose we have a parent class called Person.

```
public class Person
{
    protected String name;
    protected String address;

    /**
     * Default constructor
     */
    public Person(){
        System.out.println("Inside Person:Constructor");
        name = "";
        address = "";
    }

    /**
     * Constructor with 2 parameters
     */
    public Person( String name, String address ){
        this.name = name;
        this.address = address;
    }

    /**
     * Accessor methods
     */
    public String getName(){
        return name;
    }

    public String getAddress(){
        return address;
    }

    public void setName( String name ){
        this.name = name;
    }

    public void setAddress( String add ){
        this.address = add;
    }
}
```

Notice that, the attributes name and address are declared as **protected**. The reason we did this is that, we want these attributes to be accessible by the subclasses of the superclass. If we declare this as private, the subclasses won't be able to use them. Take note that all the properties of a superclass that are declared as **public, protected and default** can be accessed by its subclasses.

Now, we want to create another class named Student. Since a student is also a person, we decide to just extend the class Person, so that we can inherit all the properties and methods of the existing class Person. To do this, we write,

```
public class Student extends Person
{
    public Student(){
        System.out.println("Inside Student:Constructor");
        //some code here
    }

    // some code here
}
```

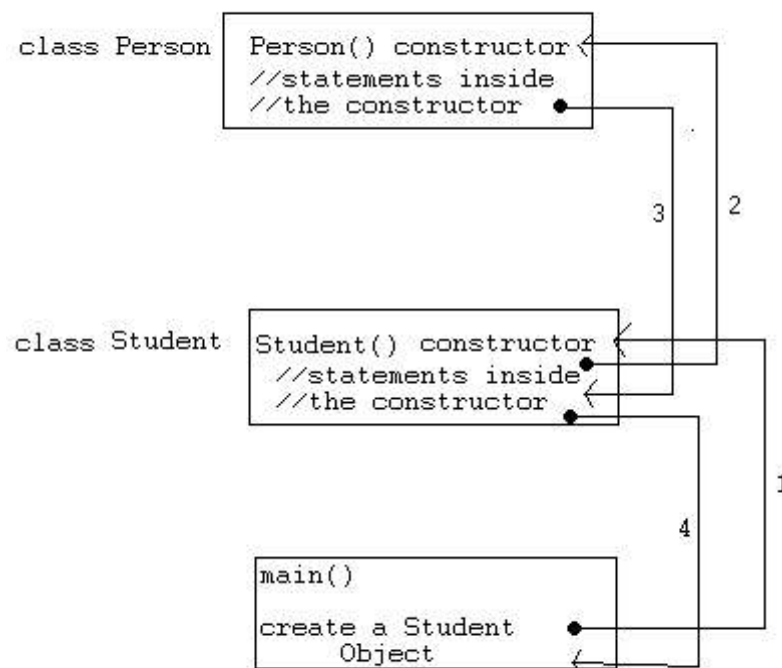
When a Student object is instantiated, the default constructor of its superclass is invoked implicitly to do the necessary initializations. After that, the statements inside the subclass are executed. To illustrate this, consider the following code,

```
public static void main( String[] args ){
    Student anna = new Student();
}
```

In the code, we create an object of class Student. The output of the program is,

```
Inside Person:Constructor
Inside Student:Constructor
```

The program flow is shown below.



11.1.2 The super keyword

A subclass can also **explicitly** call a constructor of its immediate superclass. This is done by using the **super** constructor call. A super constructor call in the constructor of a subclass will result in the execution of relevant constructor from the superclass, based on the arguments passed.

For example, given our previous example classes Person and Student, we show an example of a super constructor call. Given the following code for Student,

```
public Student(){
    super( "SomeName", "SomeAddress" );
    System.out.println("Inside Student:Constructor");
}
```

This code calls the second constructor of its immediate superclass (which is Person) and executes it. Another sample code shown below,

```
public Student(){
    super() ;
    System.out.println("Inside Student:Constructor");
}
```

This code calls the default constructor of its immediate superclass (which is Person) and executes it.

There are a few things to remember when using the super constructor call:

1. The super() call MUST OCCUR THE FIRST STATEMENT IN A CONSTRUCTOR.
2. The super() call can only be used in a constructor definition.
3. This implies that the this() construct and the super() calls CANNOT BOTH OCCUR IN THE SAME CONSTRUCTOR.

Another use of super is to refer to members of the superclass (just like the **this** reference). For example,

```
public Student()
{
    super.name = "somename";
    super.address = "some address";
}
```

11.1.3 Overriding Methods

If for some reason a derived class needs to have a different implementation of a certain method from that of the superclass, **overriding** methods could prove to be very useful. A subclass can override a method defined in its superclass by providing a new implementation for that method.

Suppose we have the following implementation for the `getName` method in the `Person` superclass,

```
public class Person
{
    :
    :
    public String getName(){
        System.out.println("Parent: getName");
        return name;
    }
    :
}
```

To override, the `getName` method in the subclass `Student`, we write,

```
public class Student extends Person
{
    :
    :
    public String getName(){
        System.out.println("Student: getName");
        return name;
    }
    :
}
```

So, when we invoke the `getName` method of an object of class `Student`, the overridden method would be called, and the output would be,

```
Student: getName
```

11.1.4 Final Methods and Final Classes

In Java, it is also possible to declare classes that can no longer be subclassed. These classes are called **final classes**. To declare a class to be final, we just add the final keyword in the class declaration. For example, if we want the class Person to be declared final, we write,

```
public final class Person
{
    //some code here
}
```

Many of the classes in the Java API are declared final to ensure that their behavior cannot be overridden. Examples of these classes are Integer, Double and String.

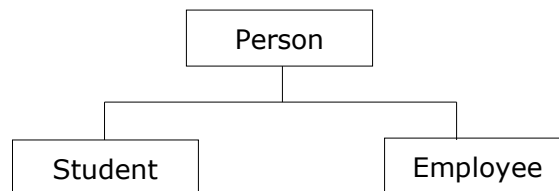
It is also possible in Java to create methods that cannot be overridden. These methods are what we call **final methods**. To declare a method to be final, we just add the final keyword in the method declaration. For example, if we want the getName method in class Person to be declared final, we write,

```
public final String getName(){
    return name;
}
```

Static methods are automatically final. This means that you cannot override them.

11.2 Polymorphism

Now, given the parent class Person and the subclass Student of our previous example, we add another subclass of Person which is Employee. Below is the class hierarchy for that,



In Java, we can create a reference that is of type superclass to an object of its subclass. For example,

```
public static main( String[] args )
{
    Person    ref;

    Student    studentObject = new Student();
    Employee    employeeObject = new Employee();

    ref = studentObject; //Person ref points to a
                        // Student object

    //some code here
}
```

Now suppose we have a `getName` method in our superclass `Person`, and we override this method in both the subclasses `Student` and `Employee`,

```
public class Person
{
    :
    public String getName(){
        System.out.println("Person Name:" + name);
        return name;
    }
    :
}

public class Student extends Person
{
    :
    public String getName(){
        System.out.println("Student Name:" + name);
        return name;
    }
    :
}

public class Employee extends Person
{
    :
    public String getName(){
        System.out.println("Employee Name:" + name);
        return name;
    }
    :
}
```


Going back to our main method, when we try to call the getName method of the reference Person ref, the getName method of the Student object will be called. Now, if we assign ref to an Employee object, the getName method of Employee will be called.

```
public static main( String[] args )
{
    Person      ref;

    Student      studentObject = new Student();
    Employee     employeeObject = new Employee();

    ref = studentObject; //Person reference points to a
                        // Student object
    String temp = ref.getName(); //getName of Student
                                //class is called
    System.out.println( temp );

    ref = employeeObject; //Person reference points to an
                        // Employee object

    String temp = ref.getName(); //getName of Employee
                                //class is called
    System.out.println( temp );
}
```

This ability of our reference to change behavior according to what object it is holding is called **polymorphism**. Polymorphism allows multiple objects of different subclasses to be treated as objects of a single superclass, while automatically selecting the proper methods to apply to a particular object based on the subclass it belongs to.

11.3 Abstract Classes

Now suppose we want to create a superclass wherein it has certain methods in it that contains some implementation, and some methods wherein we just want to be overridden by its subclasses.

For example, we want to create a superclass named `LivingThing`. This class has certain methods like `breath`, `eat`, `sleep` and `walk`. However, there are some methods in this superclass wherein we cannot generalize the behavior. Take for example, the `walk` method. Not all living things walk the same way. Take the humans for instance, we humans walk on two legs, while other living things like dogs walk on four legs. However, there are many characteristics that living things have in common, that is why we want to create a general superclass for this.

In order to do this, we can create a superclass that has some methods with implementations and others which do not. This kind of class is called an abstract class.

An **abstract class** is a class that cannot be instantiated. It often appears at the top of an object-oriented programming class hierarchy, defining the broad types of actions possible with objects of all subclasses of the class.

Those methods in the abstract classes that do not have implementation are called **abstract methods**. To create an abstract method, just write the method declaration without the body and use the `abstract` keyword. For example,

```
public abstract void someMethod();
```

Now, let's create an example abstract class.

```
public abstract class LivingThing
{
    public void breath(){
        System.out.println("Living Thing breathing...");
    }

    public void eat(){
        System.out.println("Living Thing eating...");
    }

    /**
     * abstract method walk
     * We want this method to be overridden by subclasses of
     * LivingThing
     */
    public abstract void walk();
}
```

When a class extends the `LivingThing` abstract class, it is required to override the abstract method `walk()`, or else, that subclass will also become an abstract class, and therefore cannot be instantiated. For example,

```
public class Human extends LivingThing
{
    public void walk(){
        System.out.println("Human walks...");
    }
}
```

If the class `Human` does not override the `walk` method, we would encounter the following error message,

```
Human.java:1: Human is not abstract and does not override
abstract method walk() in LivingThing
public class Human extends LivingThing
      ^
1 error
```

Coding Guidelines:

Use abstract classes to define broad types of behaviors at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class.

11.4 Interfaces

An **interface** is a special kind of block containing method signatures (and possibly constants) only. Interfaces define the signatures of a set of methods, without the body.

Interfaces define a standard and public way of specifying the behavior of classes. They allow classes, regardless of their locations in the class hierarchy, to implement common behaviors. Note that interfaces exhibit polymorphism as well, since program may call an interface method, and the proper version of that method will be executed depending on the type of object passed to the interface method call.

The main difference between an interface and an abstract class is that, in an interface, all methods do not have a body and you can only define constants. For abstract classes, it is just like ordinary class declarations, with some abstract methods.

Interfaces have no direct inherited relationship with any particular class, they are defined independently.

To create an interface, we write,

```
public interface [InterfaceName]
{
    //some methods without the body
}
```

As an example, let's create an interface that defines relationships between two objects according to the "natural order" of the objects.

```
public interface Relation
{
    public boolean isGreater( Object a, Object b);
    public boolean isLess( Object a, Object b);
    public boolean isEqual( Object a, Object b);
}
```

Now, to use the interface, we use the **implements** keyword. For example,

```
/**
 * This class defines a line segment
 */
public class Line implements Relation
{
    private double x1;
    private double x2;
    private double y1;
    private double y2;

    public Line( double    x1, double x2, double y1, double
y2 ){
        this.x1 = x1;
        this.x2 = x2;
        this.y1 = y1;
        this.y2 = y2;
    }

    public double getLength(){
        double length = Math.sqrt((x2-x1)*(x2-x1) + (y2-y1)
* (y2-y1));

        return length;
    }

    public boolean isGreater( Object a, Object b){
        double aLen = ((Line)a).getLength();
        double bLen = ((Line)b).getLength();

        return (aLen > bLen);
    }

    public boolean isLess( Object a, Object b){
        double aLen = ((Line)a).getLength();
        double bLen = ((Line)b).getLength();

        return (aLen < bLen);
    }

    public boolean isEqual( Object a, Object b){
        double aLen = ((Line)a).getLength();
        double bLen = ((Line)b).getLength();

        return (aLen == bLen);
    }
}
```

When your class tries to implement an interface, always make sure that you implement all the methods of that interface, or else, you would encounter this error,

```
Line.java:4: Line is not abstract and does not override
abstract method isGreater(java.lang.Object,java.lang.Object) in
Relation
public class Line implements Relation
      ^
1 error
```

Coding Guidelines:

Use interfaces to create the same standard method definitions in many different classes. Once a set of standard method definition is created, you can write a single method to manipulate all of the classes that implement the interface.

11.5 Exercises

11.5.1 Extending StudentRecord

In this exercise, we want to create a more specialized student record that contains additional information about a Computer Science student. Your task is to extend the StudentRecord class that was implemented in the previous lessons. Add some attributes and methods that you think are needed for a Computer Science student record. Try to override some existing methods in the superclass StudentRecord, if you really need to.

11.5.2 The Shape abstract class

Try to create an abstract class called Shape with abstract methods `getArea()` and `getName()`. Write two of its subclasses Circle and Square. You can add additional methods to its subclasses if you want to.

12 Basic Exception Handling

In this section, we are going to study a technique used in Java to handle unusual conditions that interrupt the normal operation of the program. This technique is called **exception handling**.

12.1 What are Exceptions?

An exception is an event that interrupts the normal processing flow of a program. This event is usually some error of some sort. This causes our program to terminate abnormally.

Some examples of exceptions that you might have encountered in our previous exercises are: `ArrayIndexOutOfBoundsException` exceptions, which occurs if we try to access a non-existent array element, or maybe a `NumberFormatException`, which occurs when we try to pass as a parameter a non-number in the `Integer.parseInt` method.

12.2 Handling Exceptions

To handle exceptions in Java, we use a try-catch-finally block. What we do in our programs is that we place the statements that can possibly generate an exception inside this block.

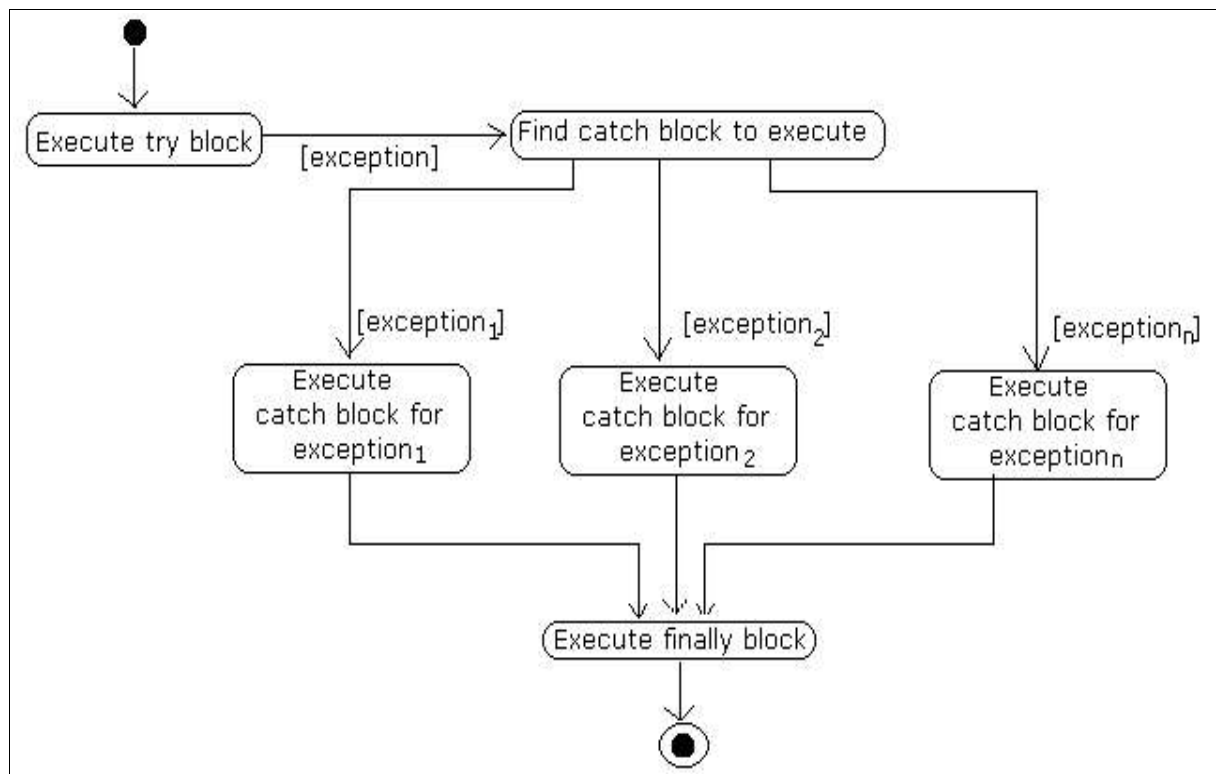
The general form of a try-catch-finally block is,

```
try{
    //write the statements that can generate an exception
    //in this block
}
catch( <exceptionType1> <varName1> ){
    //write the action your program will do if an exception
    //of a certain type occurs
}
. . .
catch( <exceptionTypen> <varNamen> ){
    //write the action your program will do if an exception
    //of a certain type occurs
}
finally{
    //add more cleanup code here
}
```

Exceptions thrown during execution of the try block can be caught and handled in a catch block. The code in the finally block is always executed.

The following are the key aspects about the syntax of the try-catch-finally construct:

- The block notation is mandatory.
- For each try block, there can be one or more catch blocks, but only one finally block.
- The catch blocks and finally blocks must always appear in conjunction with the try block, and in the above order.
- A try block must be followed by **at least** one catch block OR one finally block, or both.
- Each catch block defines an exception handle. The header of the catch block takes exactly one argument, which is the exception its block is willing to handle. The exception must be of the Throwable class or one of its subclasses.



Let's take for example a code that prints the second argument when we try to run the code using command-line arguments. Suppose, there is no checking inside your code for the number of arguments and we just access the second argument `args[1]` right away, we'll get the following exception.

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 1
    at ExceptionExample.main(ExceptionExample.java:5)
```

To prevent this from happening, we can place the code inside a try-catch block. The finally block is just optional. For this example, we won't use the finally block.

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

        try{
            System.out.println( args[1] );
        }catch( ArrayIndexOutOfBoundsException exp ){
            System.out.println("Exception caught!");
        }
    }
}
```

So when we try to run the program again without arguments, the output would be,

```
Exception caught!
```

12.3 Exercises

12.3.1 Catching Exceptions1

Given the following code:

```
public class TestExceptions{
    public static void main( String[] args ){
        for( int i=0; true; i++ ){
            System.out.println("args["+i+"]="+
                               args[i]);
        }
    }
}
```

Compile and run the TestExceptions program. The output should look like this:

```
javac TestExceptions one two three
args[0]=one
args[1]=two
args[2]=three
Exception in thread "main"
    java.lang.ArrayIndexOutOfBoundsException: 3
        at TestExceptions.main(1.java:4)
```

Modify the TestExceptions program to handle the exception. The output of the program after catching the exception should look like this:

```
javac TestExceptions one two three
args[0]=one
args[1]=two
args[2]=three
Exception caught:
    java.lang.ArrayIndexOutOfBoundsException: 3
Quiting...
```

12.3.2 Catching Exceptions 2

Chances are very good that some programs you've written before have encountered exceptions. Since you didn't catch the exceptions, they simply halted the execution of your code. Go back to those programs and implement exception handling.

Appendix A : Java and Netbeans Installation

In this section, we will discuss on how to install Java and Netbeans in your system (Redhat Linux 9.0/Windows XP). If you are not provided with the Java 1.5 and Netbeans 4.0 installers by your instructor, you can download a copy of the installers from the Sun Microsystems website (<http://java.sun.com/j2se/1.5.0/download.jsp> for Java and <http://www.netbeans.org/downloads/> for Netbeans). Before starting with the installation, copy the installers in your hard disk first.

For Linux:

Create a folder under the /usr directory and name it java. You will now have a /usr/java directory, and copy all the installers inside this folder.

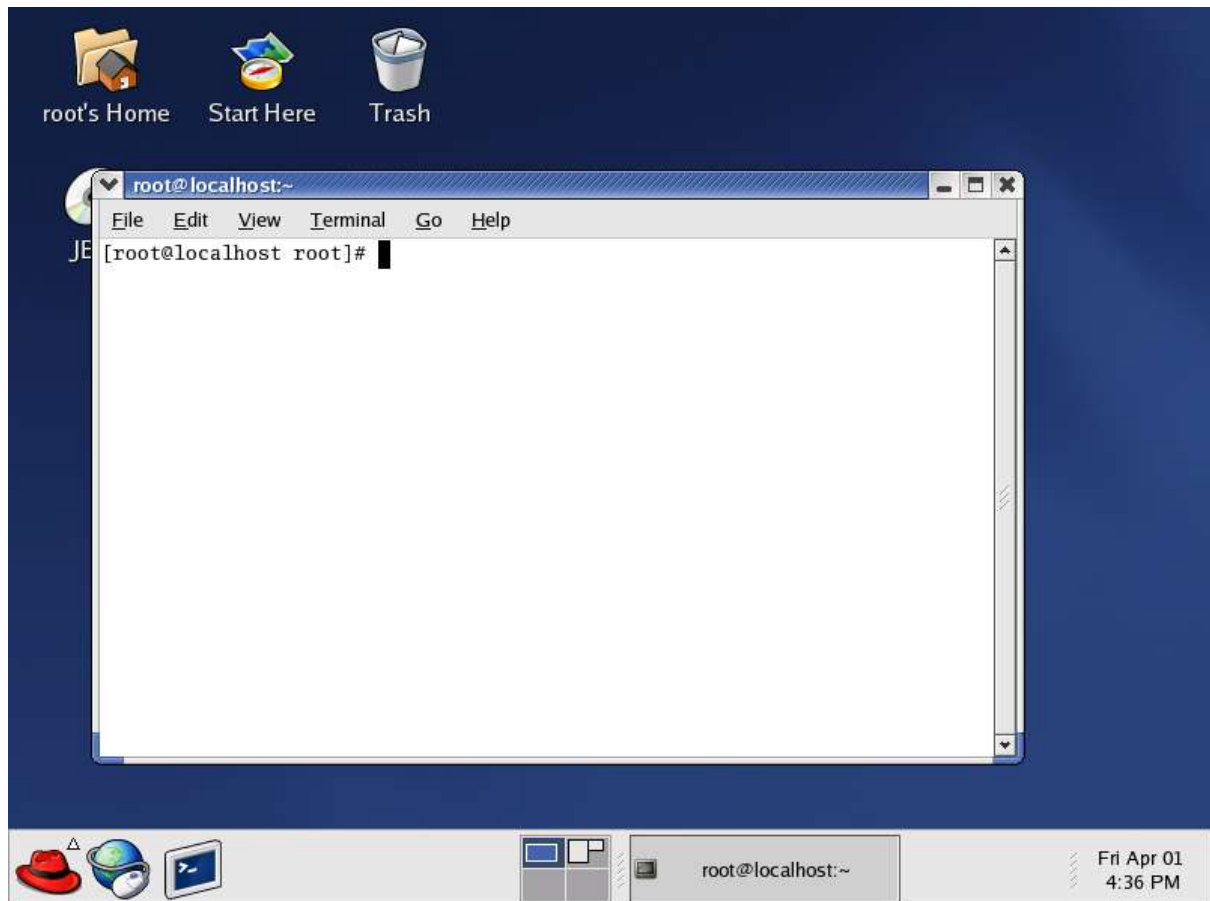
For Windows:

Just copy the installers in any temporary directory.

Installing Java in Linux

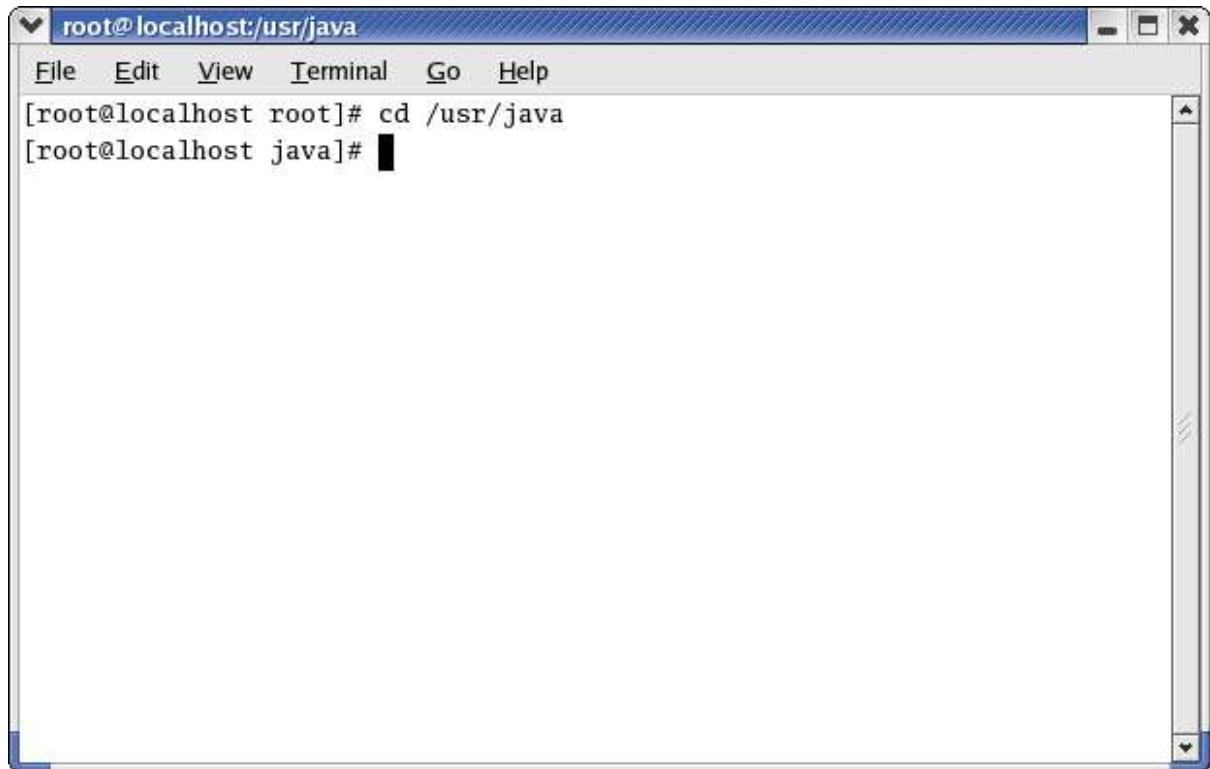
Step 1: Run Terminal

To start Terminal, click on Menu-> System Tools-> Terminal



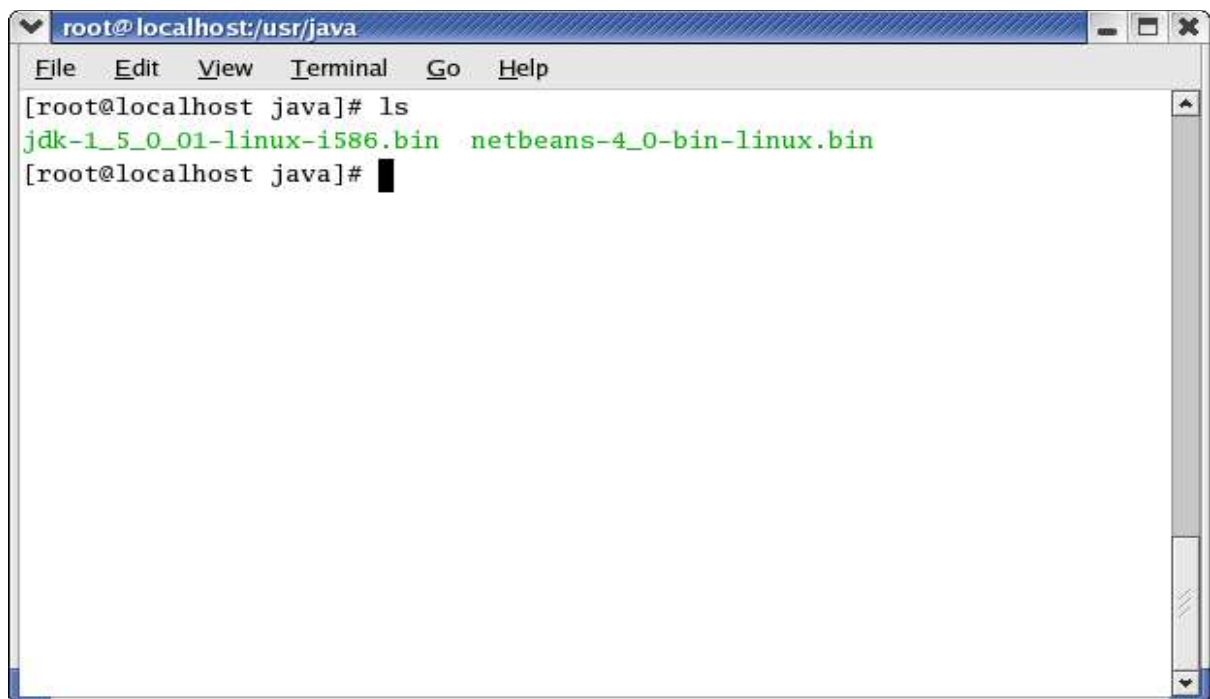
Step 2: Go to the folder where you copied the Java installer.

To go to the folder, type: `cd /usr/java`



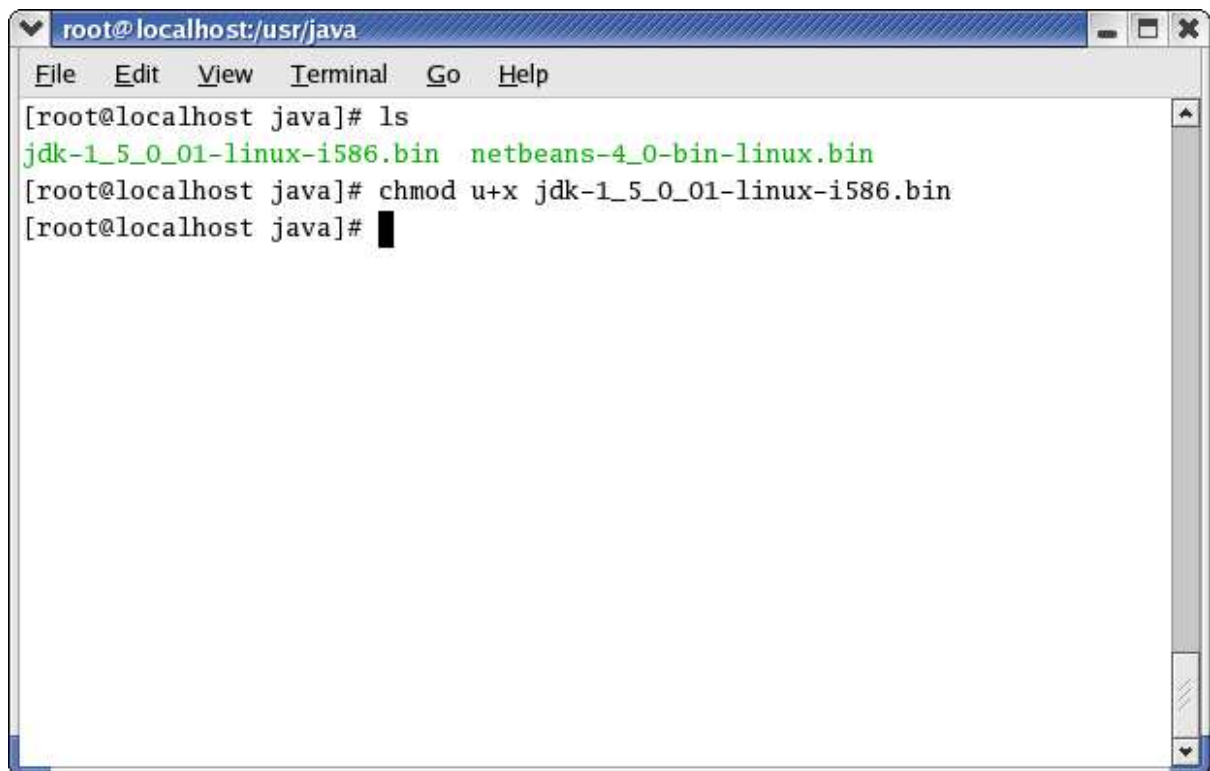
```
root@localhost:/usr/java
File Edit View Terminal Go Help
[root@localhost root]# cd /usr/java
[root@localhost java]#
```

To make sure that all installers you need are already in the folder, type: `ls`
The `ls` (list) command will list all the files inside your directory.



```
root@localhost:/usr/java
File Edit View Terminal Go Help
[root@localhost java]# ls
jdk-1_5_0_01-linux-i586.bin  netbeans-4_0-bin-linux.bin
[root@localhost java]#
```

Make your installer file executable by using the `chmod` command. To do this, type:
`chmod u+x jdk-1_5_0_01-linux-i586.bin`

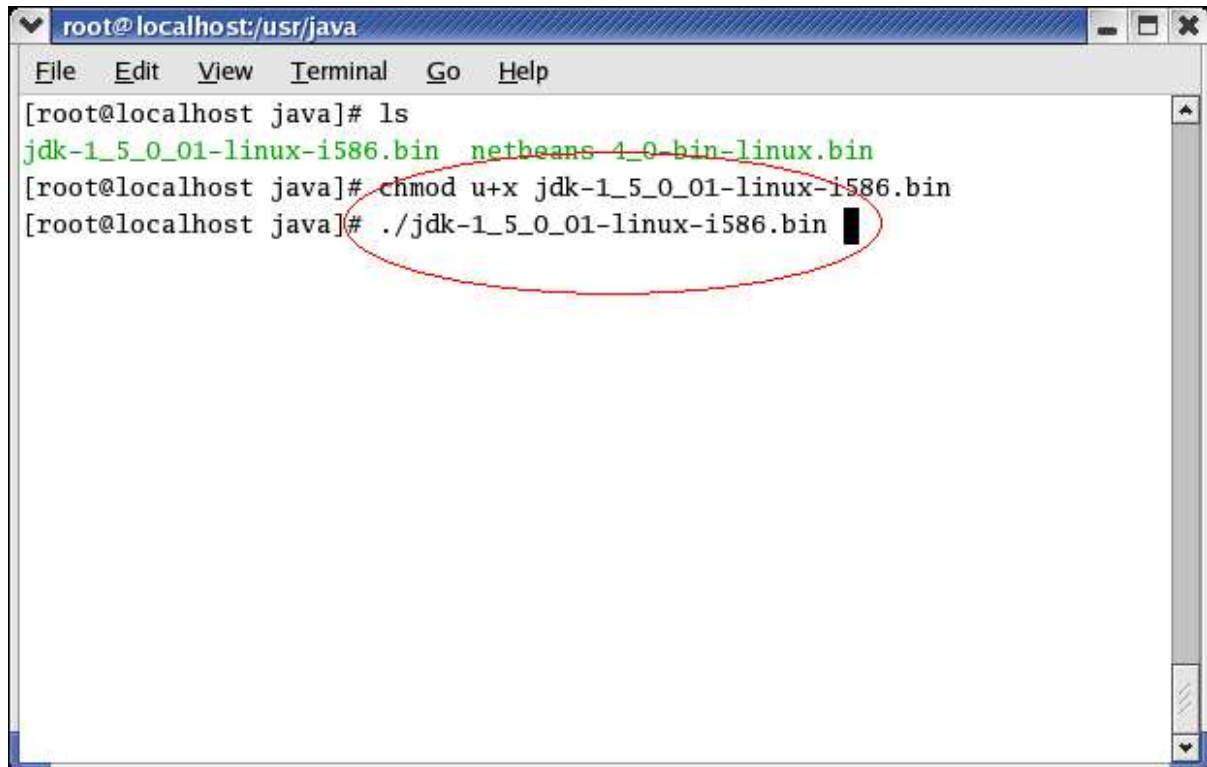
A terminal window titled 'root@localhost:/usr/java' with a menu bar containing 'File', 'Edit', 'View', 'Terminal', 'Go', and 'Help'. The terminal shows the following commands and output:

```
[root@localhost java]# ls
jdk-1_5_0_01-linux-i586.bin  netbeans-4_0-bin-linux.bin
[root@localhost java]# chmod u+x jdk-1_5_0_01-linux-i586.bin
[root@localhost java]#
```

The file names in the `ls` output are highlighted in green. A black cursor is visible on the line following the `chmod` command.

Step 3: Run the installer

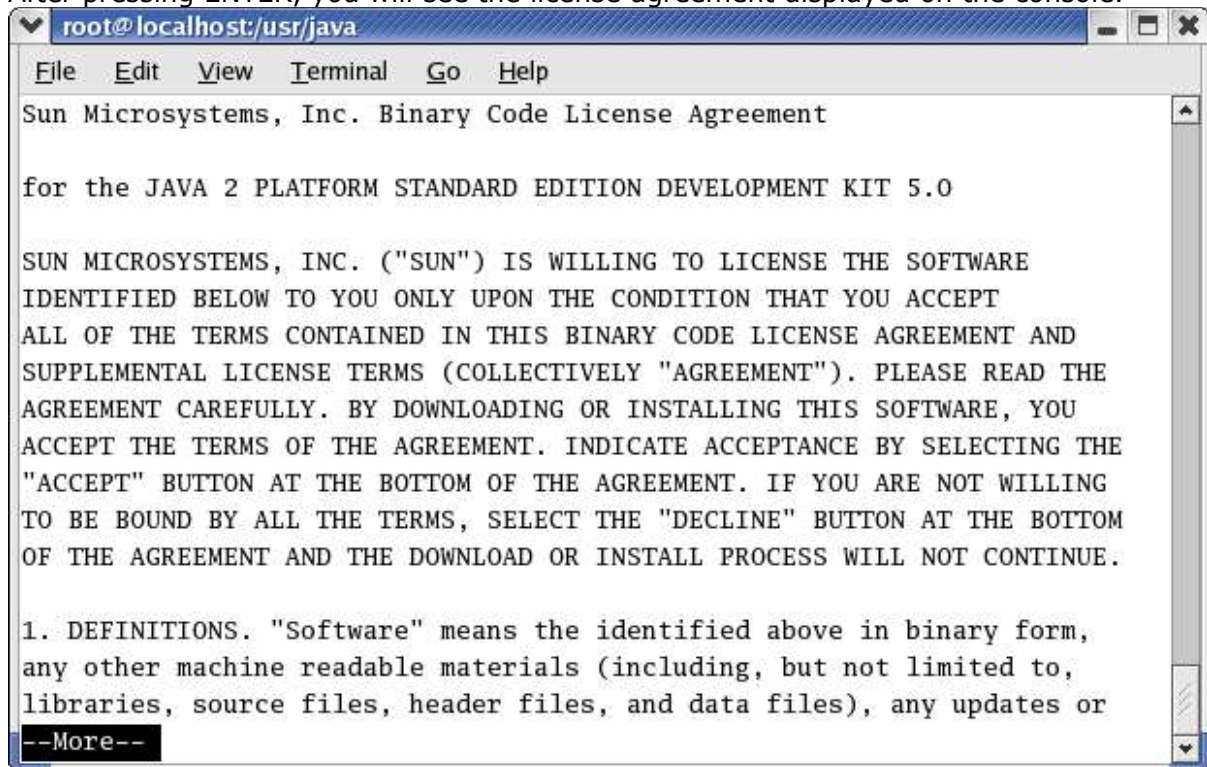
To run the installer, just type: `./jdk-1_5_0_01-linux-i586.bin`

A terminal window titled 'root@localhost:/usr/java' with a menu bar (File, Edit, View, Terminal, Go, Help). The terminal shows the following commands and output:

```
[root@localhost java]# ls
jdk-1_5_0_01-linux-i586.bin  netbeans-4_0-bin-linux.bin
[root@localhost java]# chmod u+x jdk-1_5_0_01-linux-i586.bin
[root@localhost java]# ./jdk-1_5_0_01-linux-i586.bin
```

The last command and its cursor are circled in red.

After pressing ENTER, you will see the license agreement displayed on the console.

A terminal window titled 'root@localhost:/usr/java' with a menu bar (File, Edit, View, Terminal, Go, Help). It displays the 'Sun Microsystems, Inc. Binary Code License Agreement' for the 'JAVA 2 PLATFORM STANDARD EDITION DEVELOPMENT KIT 5.0'. The text of the agreement is shown in all caps. At the bottom, it says '1. DEFINITIONS. "Software" means the identified above in binary form, any other machine readable materials (including, but not limited to, libraries, source files, header files, and data files), any updates or' followed by a black box containing '--More--'.

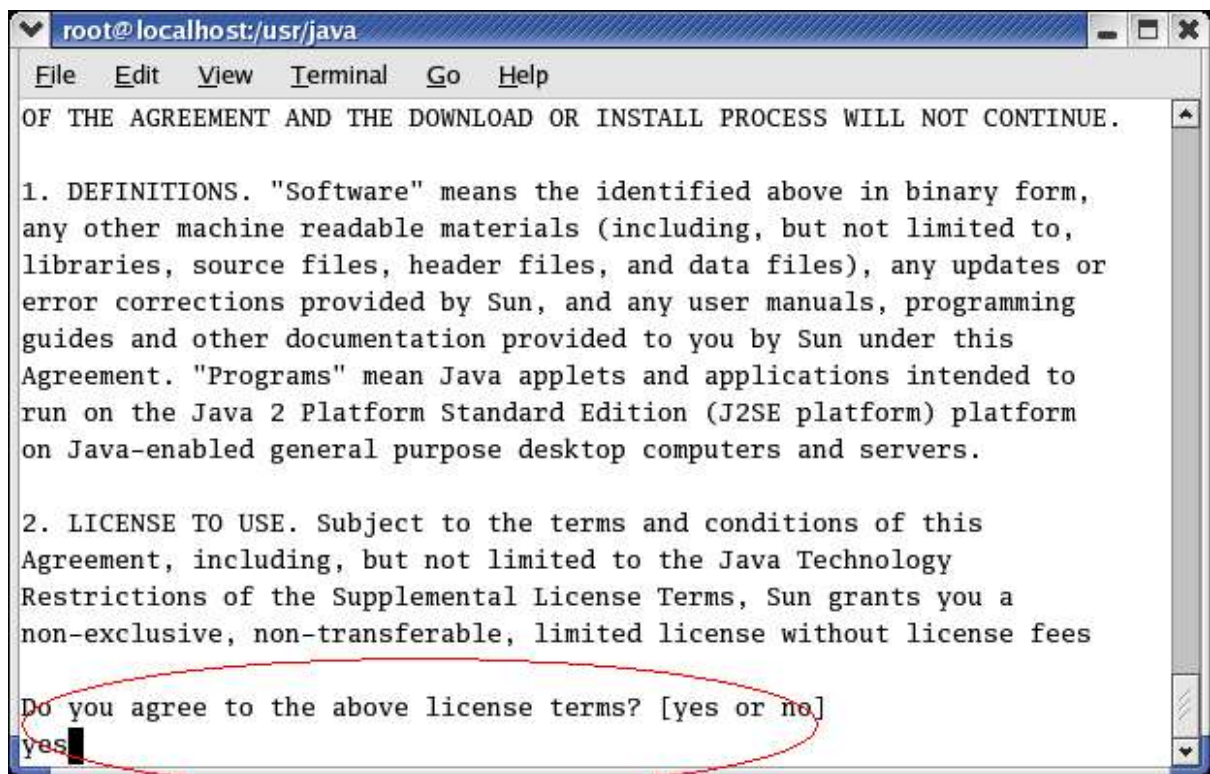
```
root@localhost:/usr/java
Sun Microsystems, Inc. Binary Code License Agreement

for the JAVA 2 PLATFORM STANDARD EDITION DEVELOPMENT KIT 5.0

SUN MICROSYSTEMS, INC. ("SUN") IS WILLING TO LICENSE THE SOFTWARE
IDENTIFIED BELOW TO YOU ONLY UPON THE CONDITION THAT YOU ACCEPT
ALL OF THE TERMS CONTAINED IN THIS BINARY CODE LICENSE AGREEMENT AND
SUPPLEMENTAL LICENSE TERMS (COLLECTIVELY "AGREEMENT"). PLEASE READ THE
AGREEMENT CAREFULLY. BY DOWNLOADING OR INSTALLING THIS SOFTWARE, YOU
ACCEPT THE TERMS OF THE AGREEMENT. INDICATE ACCEPTANCE BY SELECTING THE
"ACCEPT" BUTTON AT THE BOTTOM OF THE AGREEMENT. IF YOU ARE NOT WILLING
TO BE BOUND BY ALL THE TERMS, SELECT THE "DECLINE" BUTTON AT THE BOTTOM
OF THE AGREEMENT AND THE DOWNLOAD OR INSTALL PROCESS WILL NOT CONTINUE.

1. DEFINITIONS. "Software" means the identified above in binary form,
any other machine readable materials (including, but not limited to,
libraries, source files, header files, and data files), any updates or
--More--
```


Just press enter, until you see the question: Do you agree to the above license terms? [yes or no]. Just type: yes, and press ENTER.



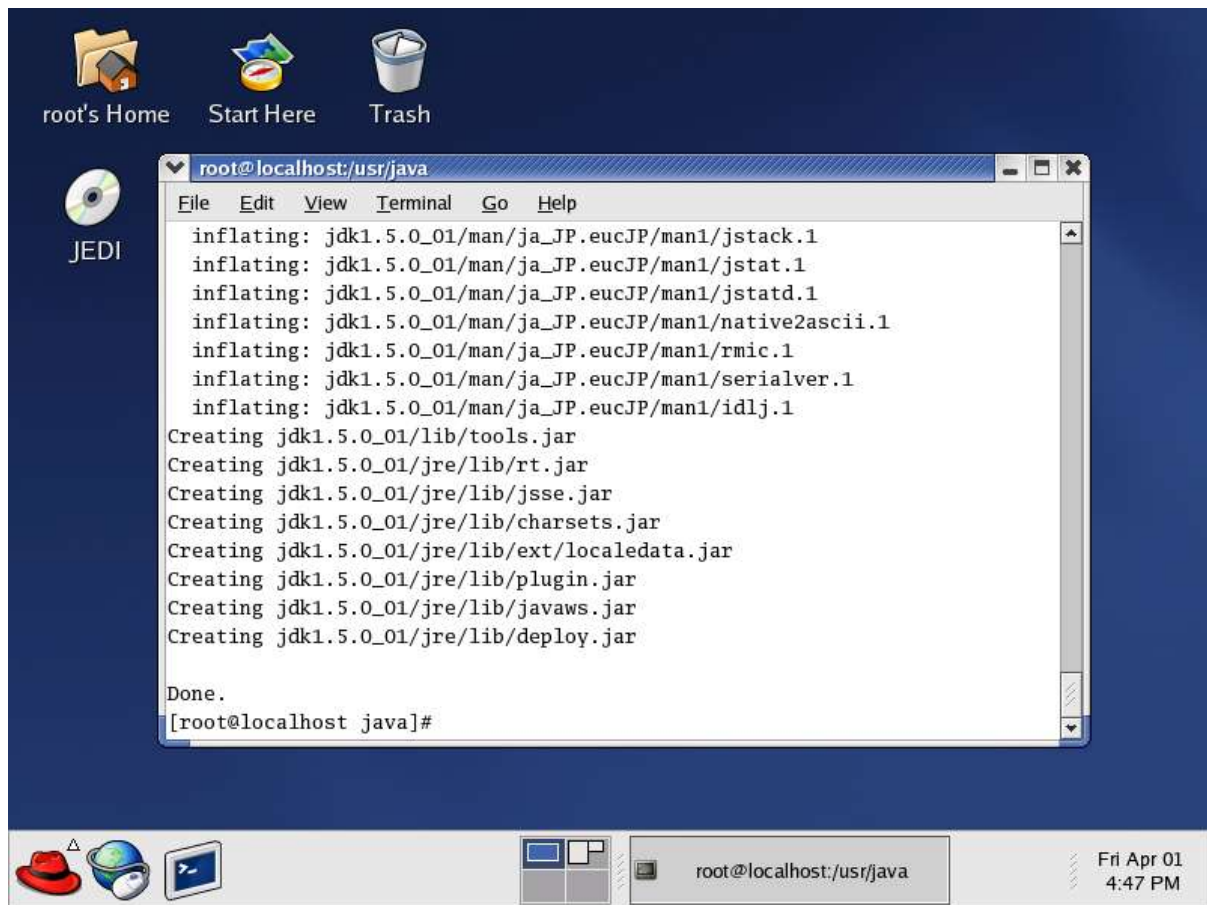
```
root@localhost:/usr/java
File Edit View Terminal Go Help
OF THE AGREEMENT AND THE DOWNLOAD OR INSTALL PROCESS WILL NOT CONTINUE.

1. DEFINITIONS. "Software" means the identified above in binary form,
any other machine readable materials (including, but not limited to,
libraries, source files, header files, and data files), any updates or
error corrections provided by Sun, and any user manuals, programming
guides and other documentation provided to you by Sun under this
Agreement. "Programs" mean Java applets and applications intended to
run on the Java 2 Platform Standard Edition (J2SE platform) platform
on Java-enabled general purpose desktop computers and servers.

2. LICENSE TO USE. Subject to the terms and conditions of this
Agreement, including, but not limited to the Java Technology
Restrictions of the Supplemental License Terms, Sun grants you a
non-exclusive, non-transferable, limited license without license fees

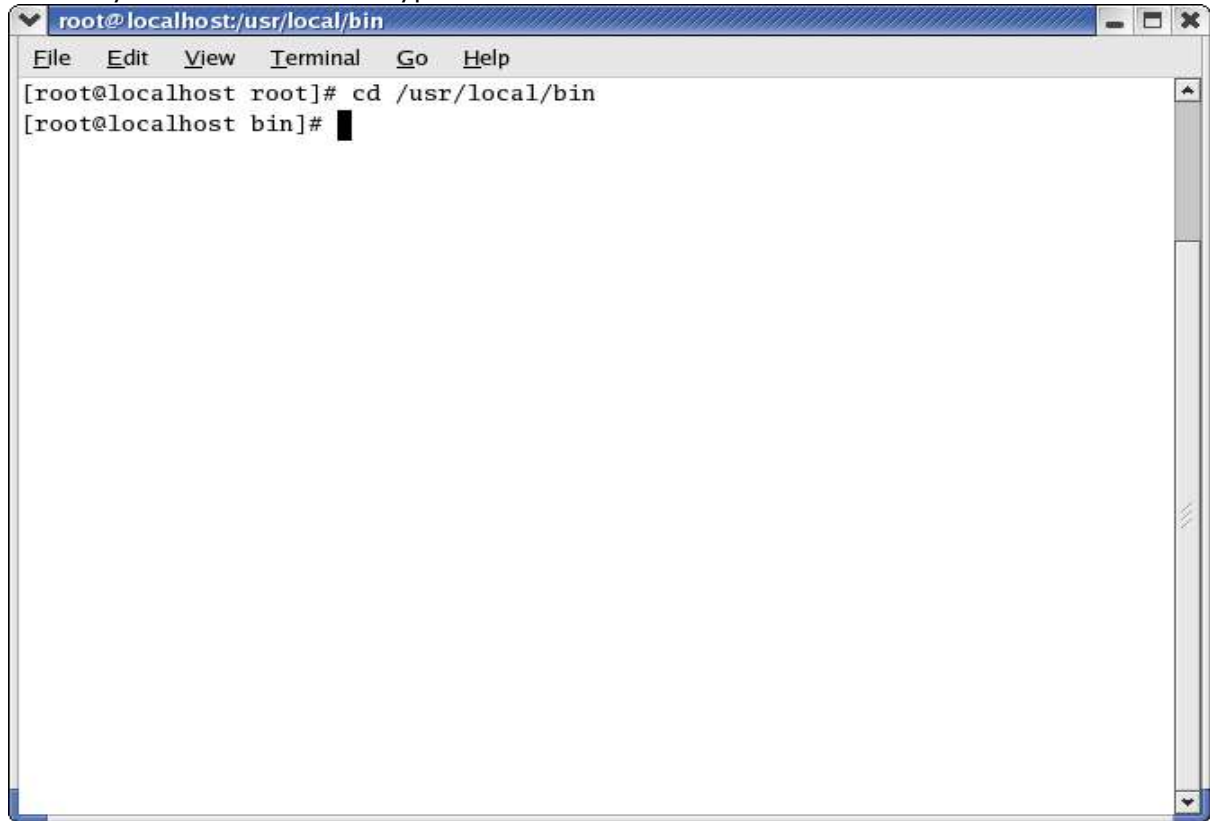
Do you agree to the above license terms? [yes or no]
yes
```

Just wait for the installer to finish unpacking all its contents and installing java.



Step 4: Creating symbolic links

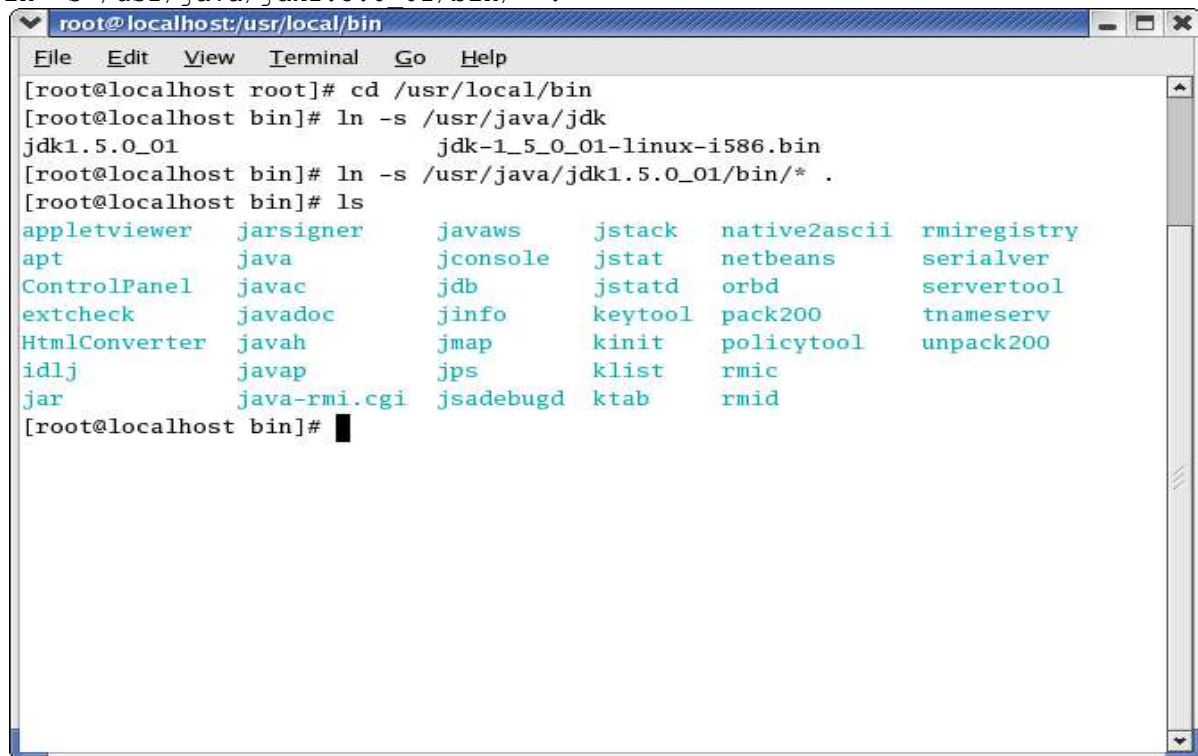
In order to run java commands anywhere, we need to create symbolic links for all the commands in JDK inside the `/usr/local/bin` directory. To do this, go to the directory `/usr/local/bin`. Type: `cd /usr/local/bin`

A screenshot of a terminal window. The title bar at the top reads "root@localhost:/usr/local/bin". Below the title bar is a menu bar with the options "File", "Edit", "View", "Terminal", "Go", and "Help". The terminal area shows the command prompt "[root@localhost root]#" followed by the command "cd /usr/local/bin". The prompt then changes to "[root@localhost bin]#" with a black cursor character at the end. The terminal window has a scroll bar on the right side.

```
root@localhost:/usr/local/bin
File Edit View Terminal Go Help
[root@localhost root]# cd /usr/local/bin
[root@localhost bin]#
```

To make the symbolic links to the commands, type:

```
ln -s /usr/java/jdk1.5.0_01/bin/* .
```

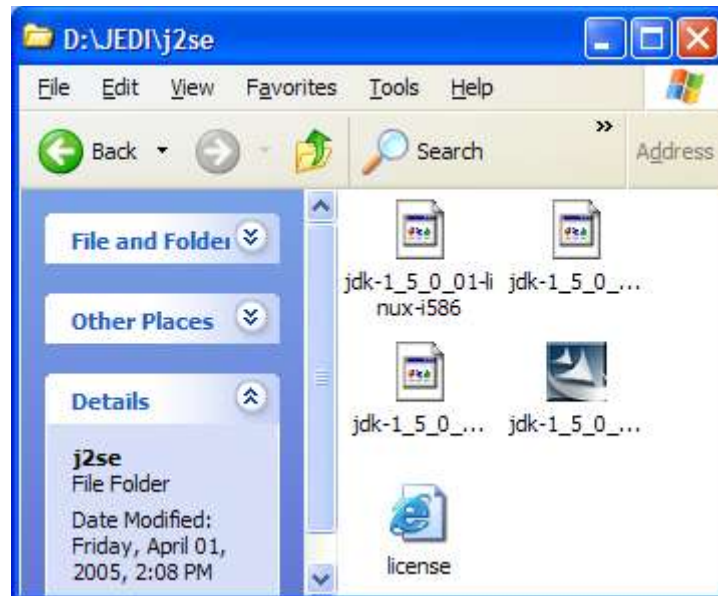


A terminal window titled "root@localhost:/usr/local/bin" with a menu bar (File, Edit, View, Terminal, Go, Help). The terminal shows the following commands and output:

```
[root@localhost root]# cd /usr/local/bin
[root@localhost bin]# ln -s /usr/java/jdk
jdk1.5.0_01                               jdk-1_5_0_01-linux-i586.bin
[root@localhost bin]# ln -s /usr/java/jdk1.5.0_01/bin/* .
[root@localhost bin]# ls
appletviewer  jarsigner      javaws          jstack          native2ascii   rmiregistry
apt           java           jconsole        jstat           netbeans        serialver
ControlPanel  javac          jdbc            jstatd          orbd            servertool
extcheck      javadoc        jinfo           keytool         pack200         tnameserv
HtmlConverter javah          jmap            kinit           policytool      unpack200
idlj          javap          jps             klist           rmic
jar           java-rmi.cgi   jsadebugd       ktab            rmid
```

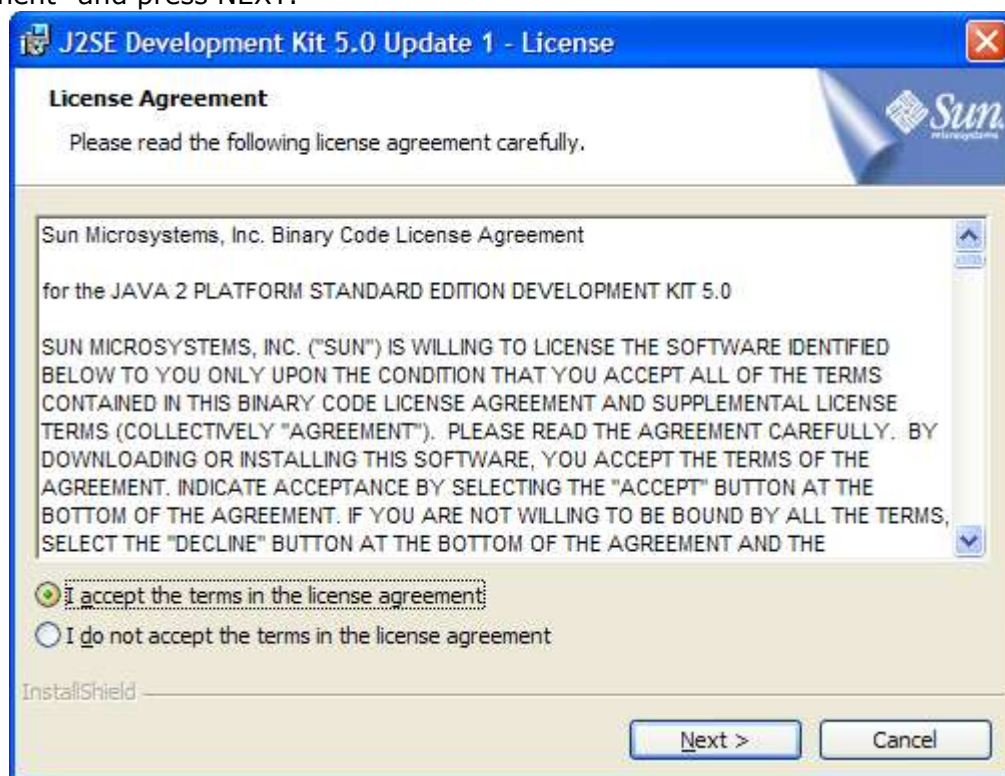
Installing Java in Windows

Step 1: Using Windows Explorer, go to the folder where your Java installer is located

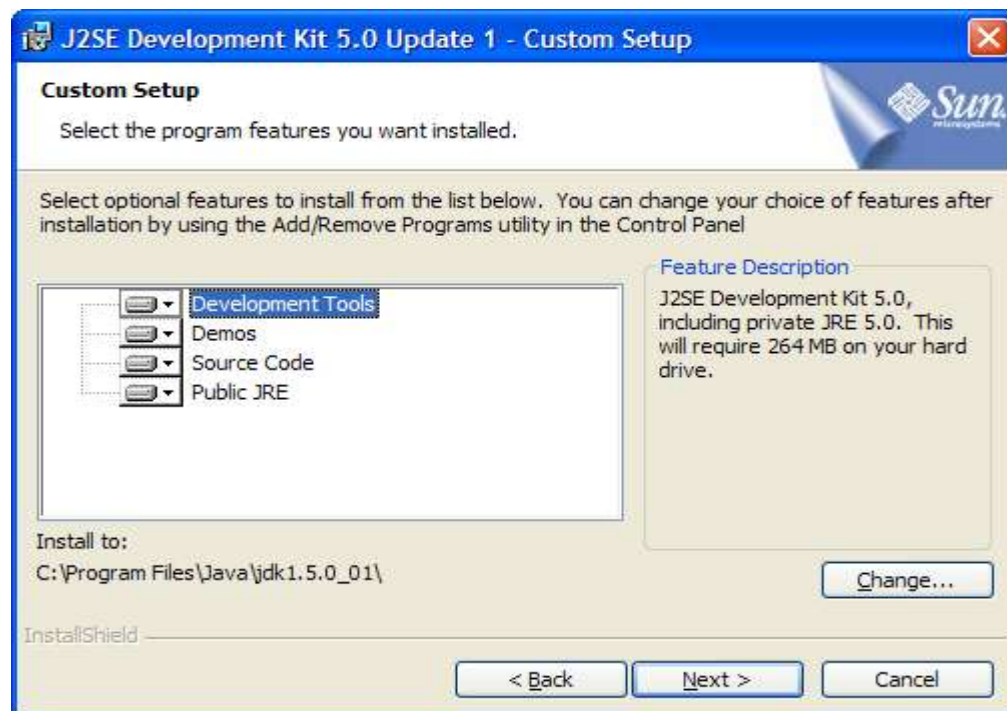


Step 2: Run the installer

To run the installer, just double-click on the installer icon. A J2SE installer dialog will then appear. Click on the radio button labeled "I accept the terms in the license agreement" and press NEXT.



Click on NEXT to continue installation.



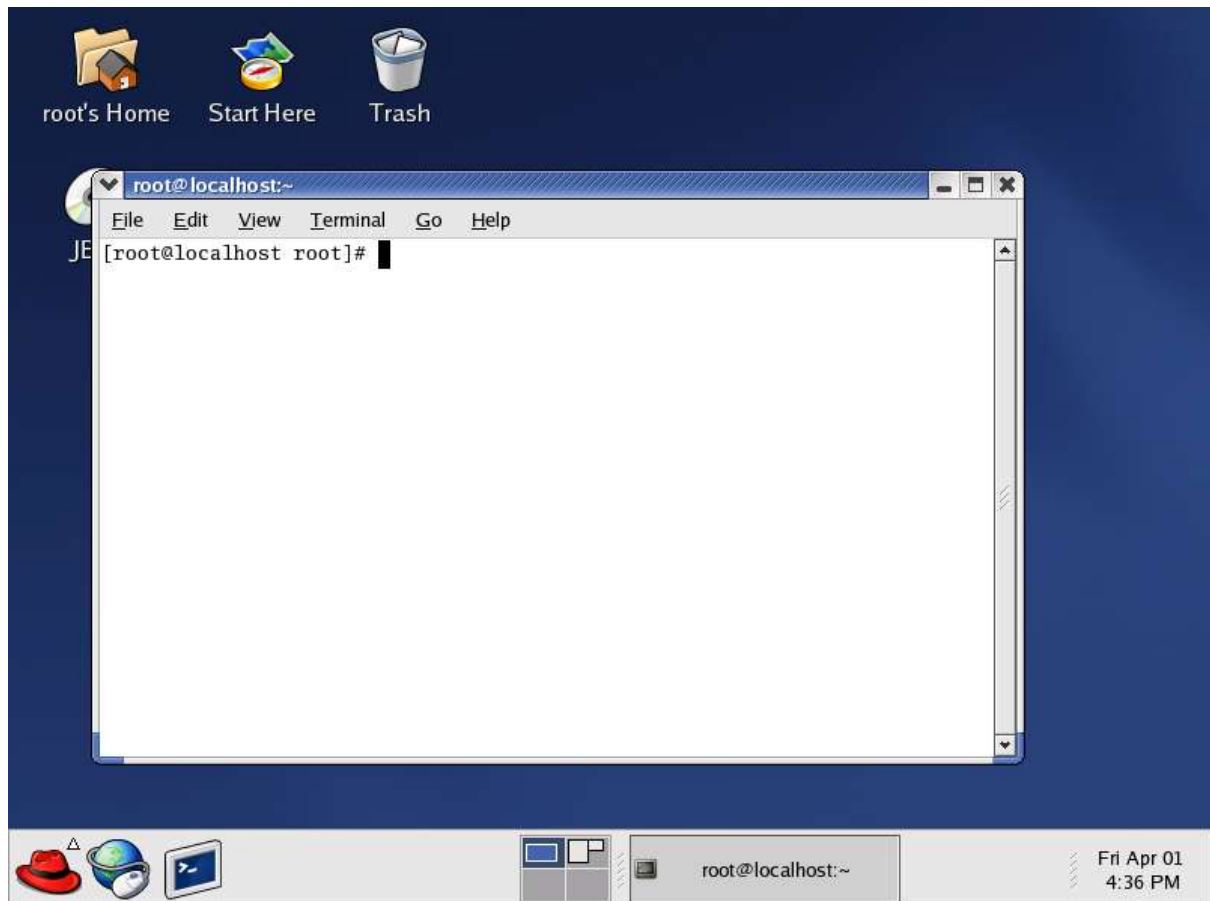
Click on FINISH to complete installation.



Installing Netbeans in Linux

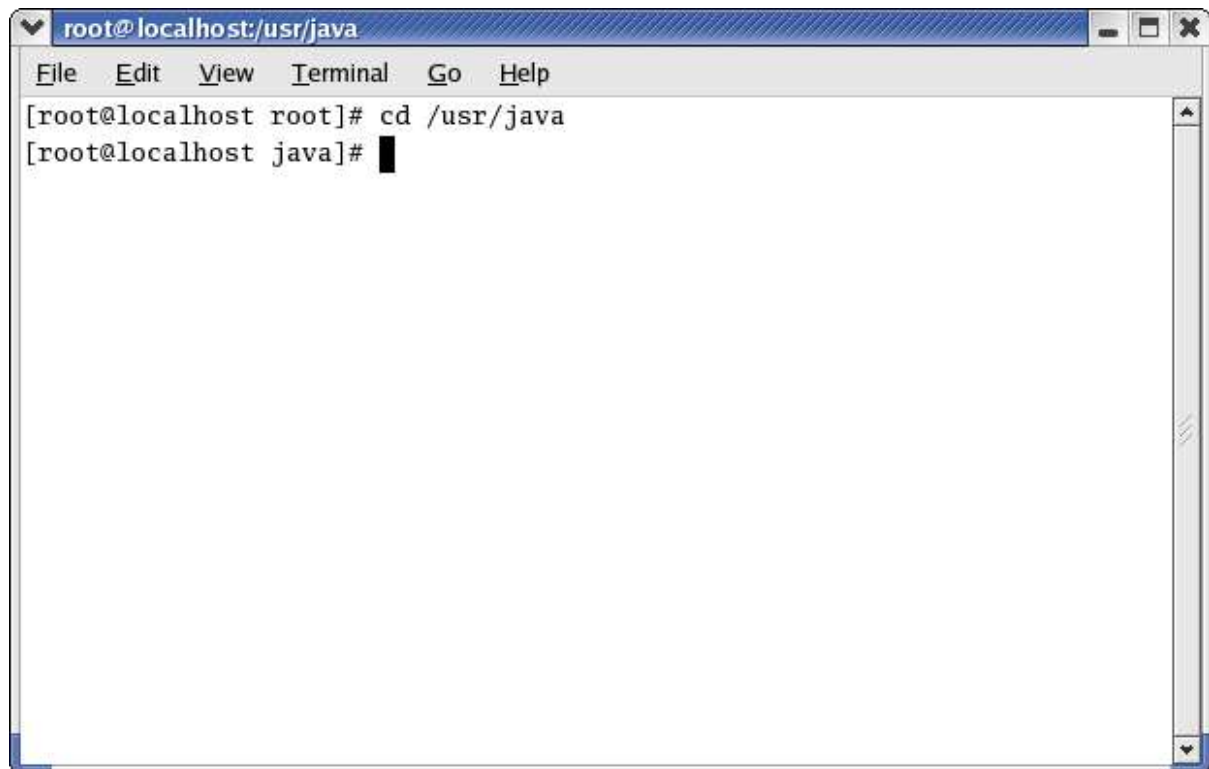
Step 1: Run Terminal

To start Terminal, click on Menu-> System Tools-> Terminal



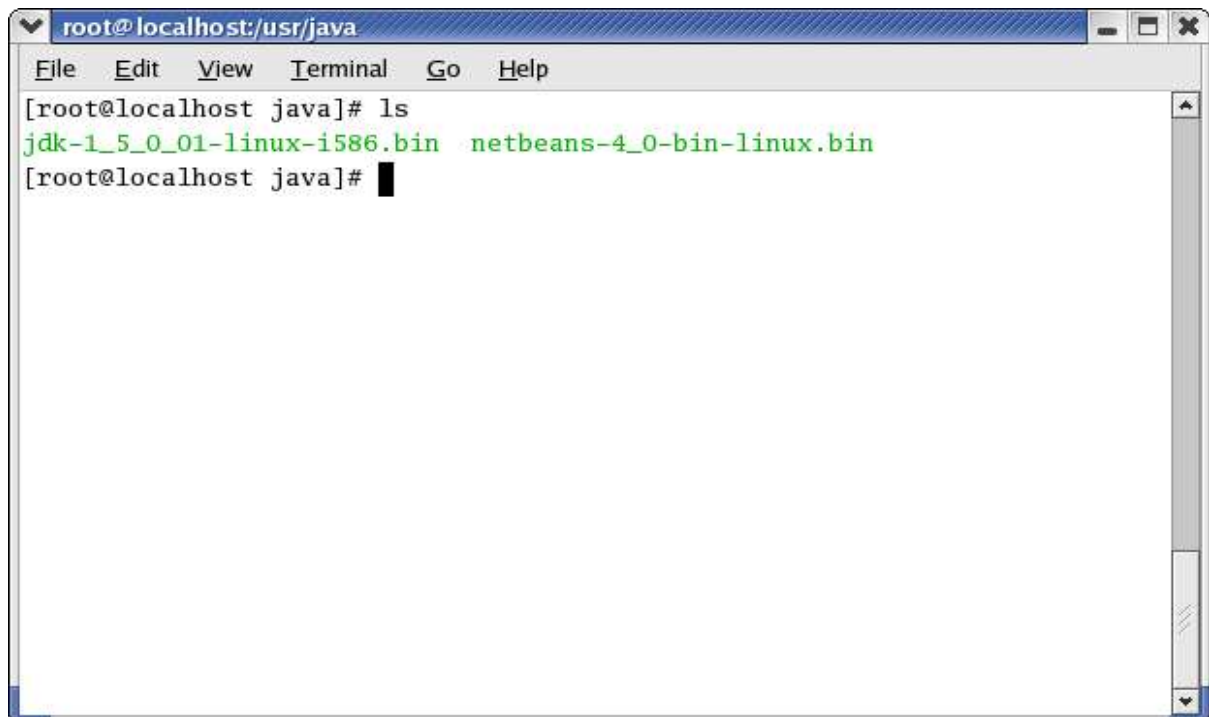
Step 2: Go to the folder where you copied the Netbeans installer.

To go to the folder, type: `cd /usr/java`



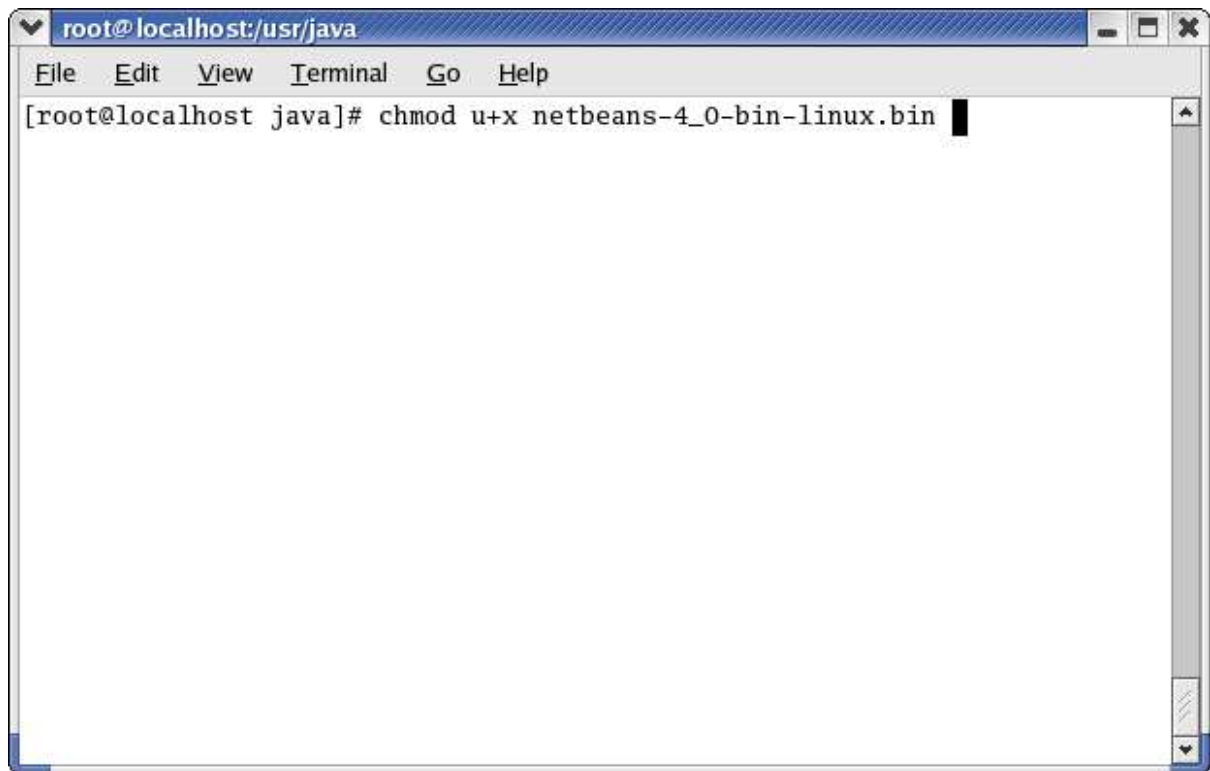
```
root@localhost:/usr/java
File Edit View Terminal Go Help
[root@localhost root]# cd /usr/java
[root@localhost java]#
```

To make sure that all installers you need are already in the folder, type: `ls`
The `ls` (list) command will list all the files inside your directory.



```
root@localhost:/usr/java
File Edit View Terminal Go Help
[root@localhost java]# ls
jdk-1_5_0_01-linux-i586.bin  netbeans-4_0-bin-linux.bin
[root@localhost java]#
```


Make your installer file executable by using the `chmod` command. To do this, type:
`chmod u+x netbeans-4_0-bin-linux.bin`



```
root@localhost:/usr/java
File Edit View Terminal Go Help
[root@localhost java]# chmod u+x netbeans-4_0-bin-linux.bin
```

Step 3: Run Installer

To run the netbeans installer, type: `./netbeans-4_0-bin-linux.bin`



```
root@localhost:/usr/java
File Edit View Terminal Go Help
[root@localhost java]# chmod u+x netbeans-4_0-bin-linux.bin
[root@localhost java]# ./netbeans-4_0-bin-linux.bin
InstallShield Wizard

Initializing InstallShield Wizard...

Searching for Java(tm) Virtual Machine...
.....
.....
```

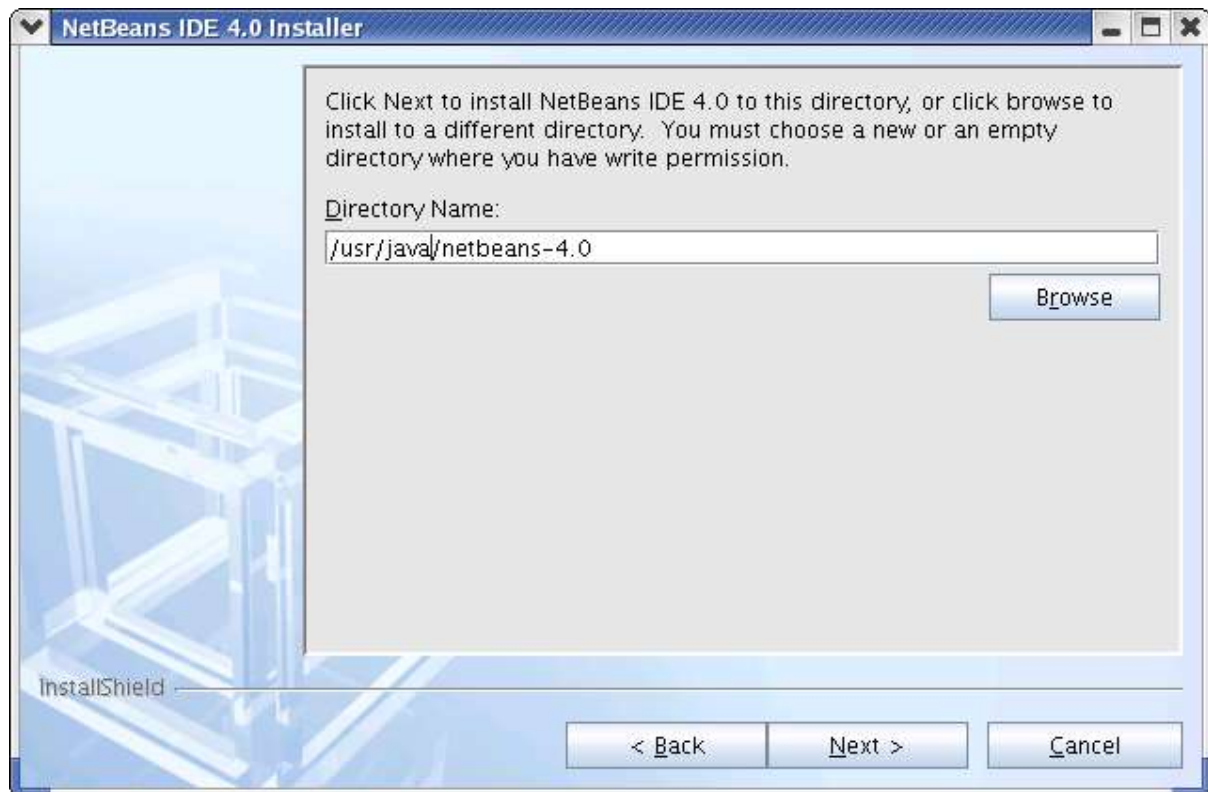
A netbeans 4.0 Installer dialog will then appear. Click on NEXT.



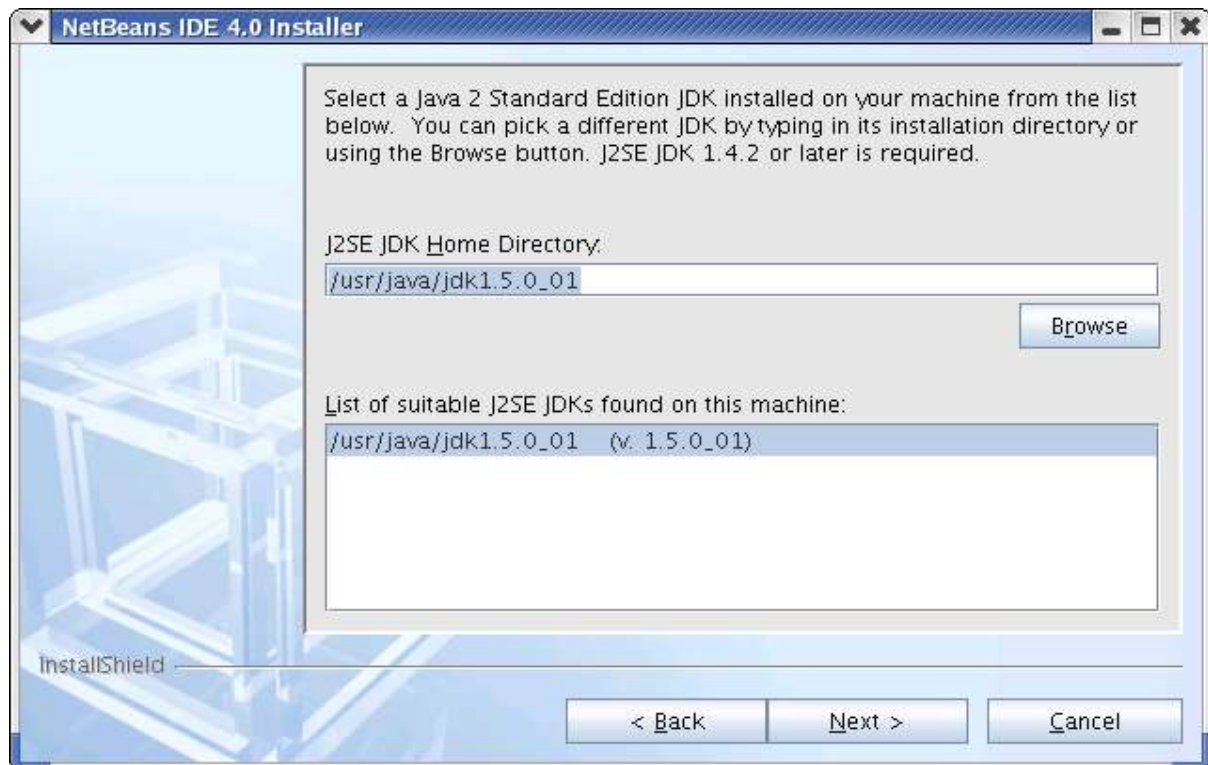
Click on the radio button that says "I accept the terms in the license agreement". And then click on NEXT.



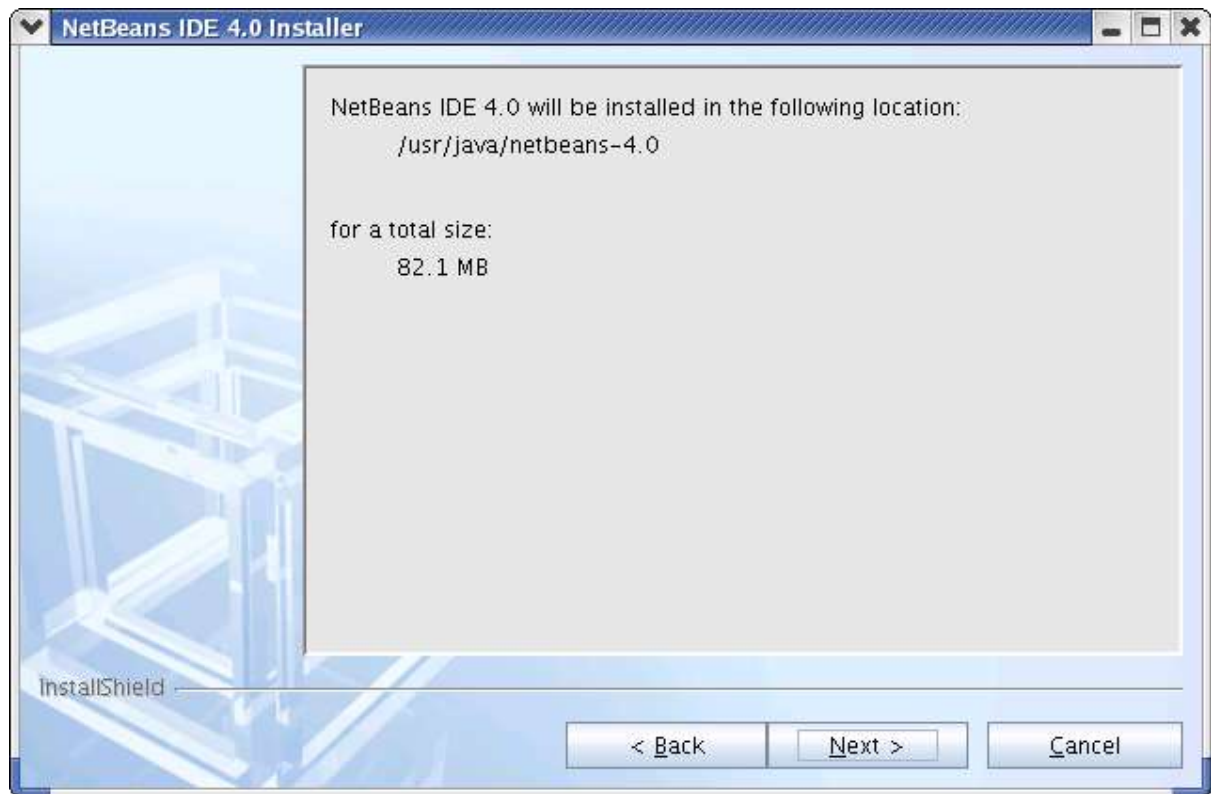
For the directory name, change it to: `/usr/java/netbeans-4.0`, then click on NEXT.



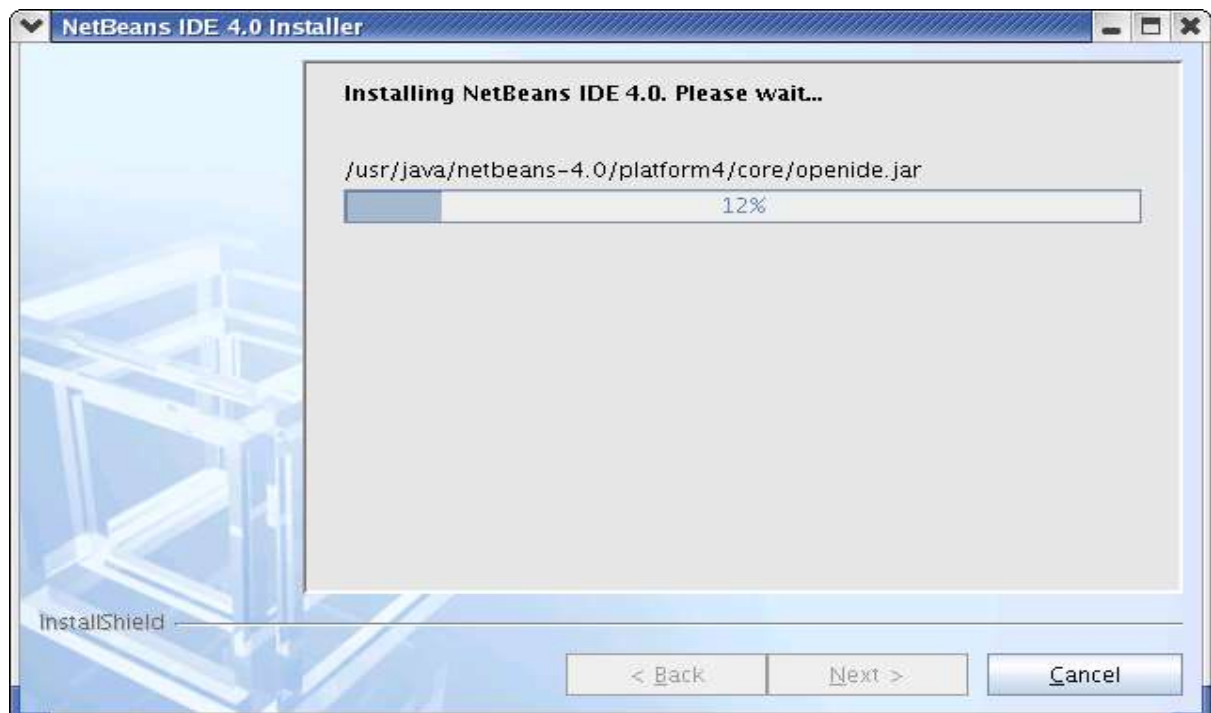
For the JDK directory, choose `/usr/java/jdk1.5.0_01`, and then click on NEXT.



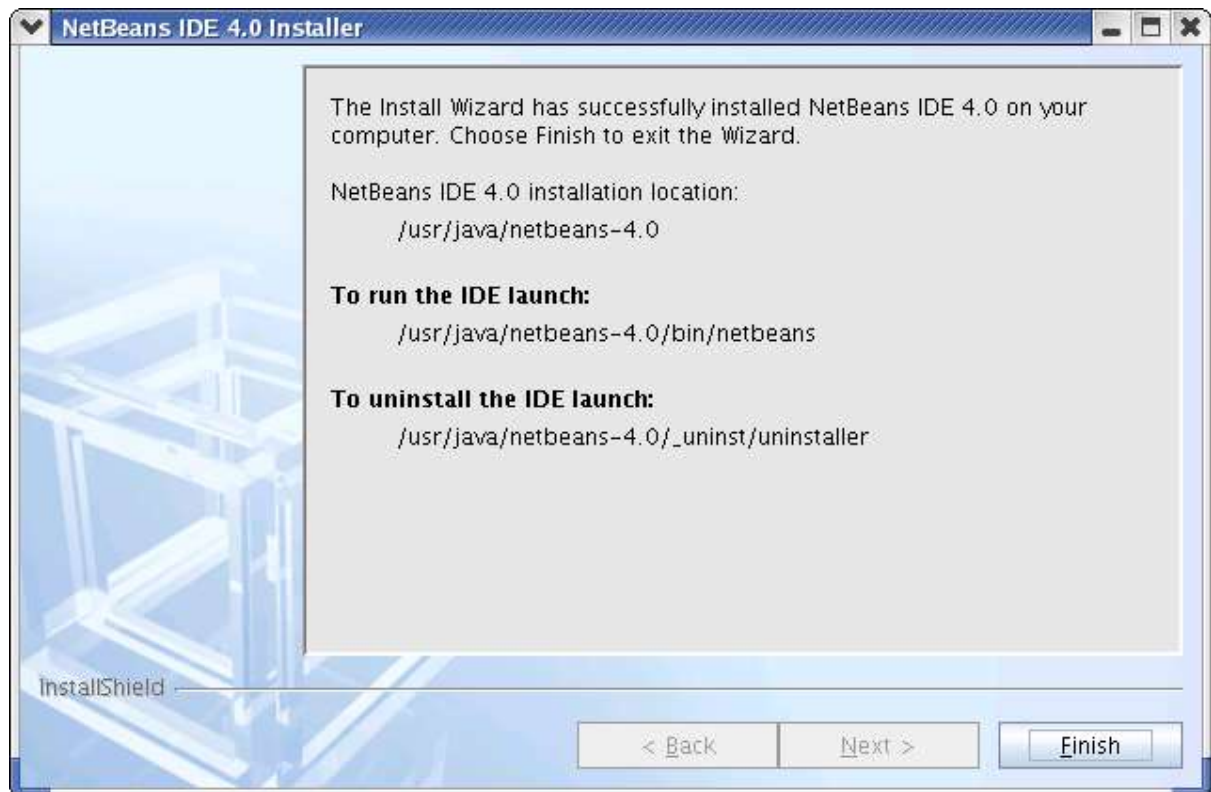
The next dialog just shows information about Netbeans that you will install. Just click again on NEXT.



Now, just wait for netbeans to finish its installation.

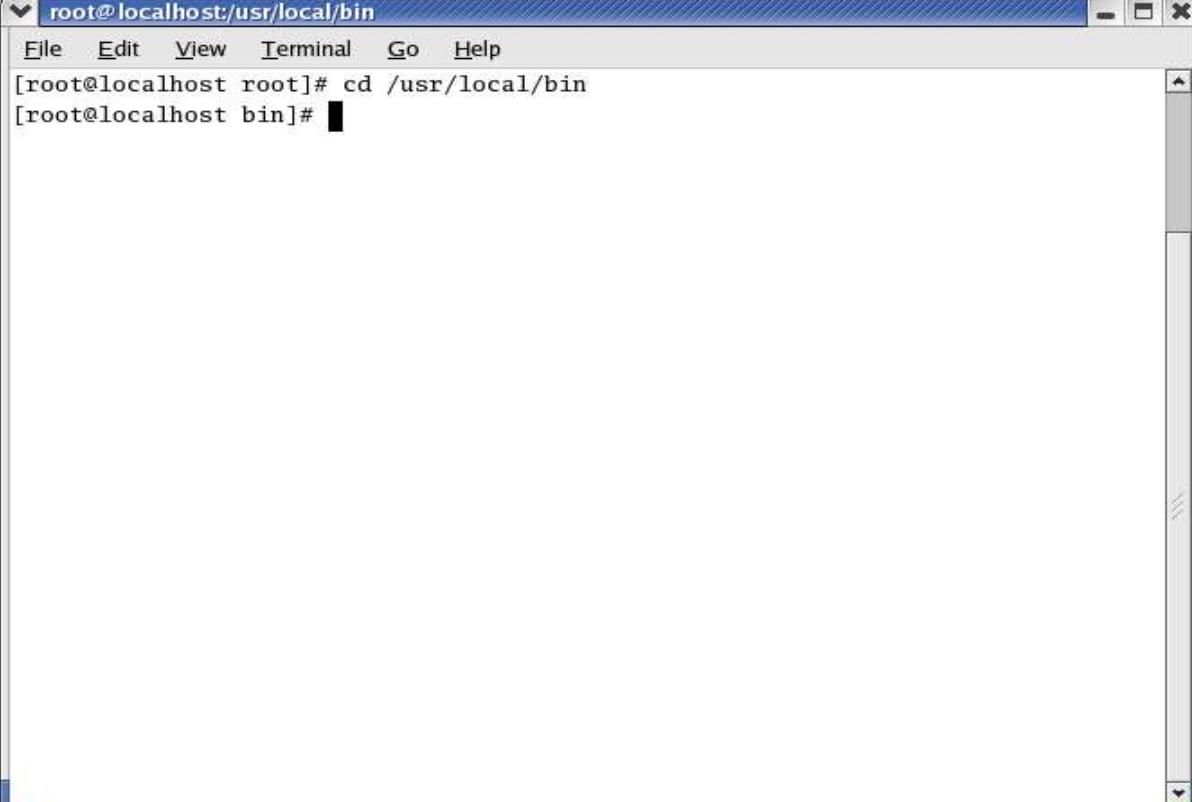


Click on FINISH to complete the installation.



Step 4: Creating symbolic links

In order to run netbeans anywhere, we need to create symbolic link for it. To do this, go first to the directory: `/usr/local/bin`. Type: `cd /usr/local/bin`



```
root@localhost:/usr/local/bin
File Edit View Terminal Go Help
[root@localhost root]# cd /usr/local/bin
[root@localhost bin]#
```

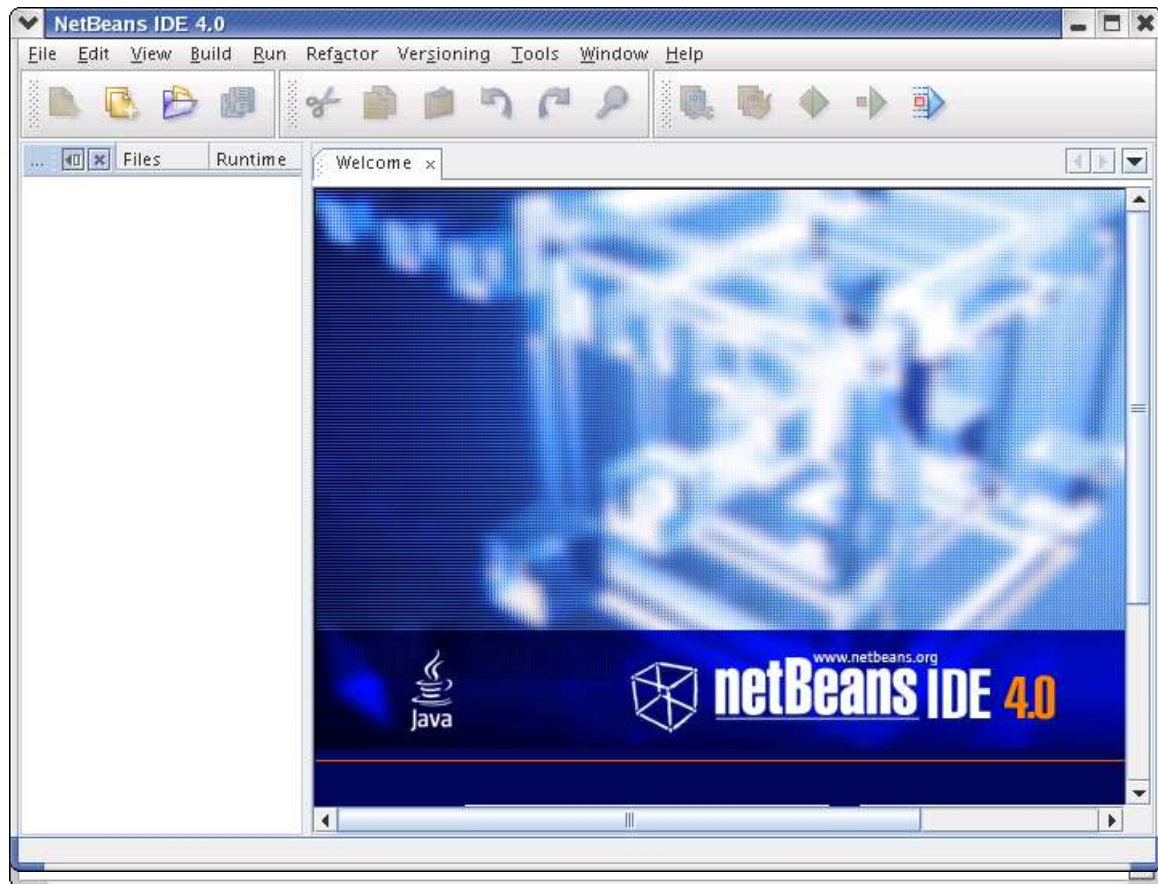
Make a symbolic link to the netbeans executable by typing:

`ln -s /usr/java/netbeans-4.0/bin/netbeans .`



```
root@localhost:/usr/local/bin
File Edit View Terminal Go Help
[root@localhost netbeans-4.0]# cd /usr/local/bin/
[root@localhost bin]# ln -s /usr/java/netbeans-4.0/bin/netbeans .
[root@localhost bin]#
```

Now, you can run netbeans in any directory by typing: `netbeans &`



Installing Netbeans in Windows

Step 1: Using Windows Explorer, go to the folder where your Netbeans installer is located

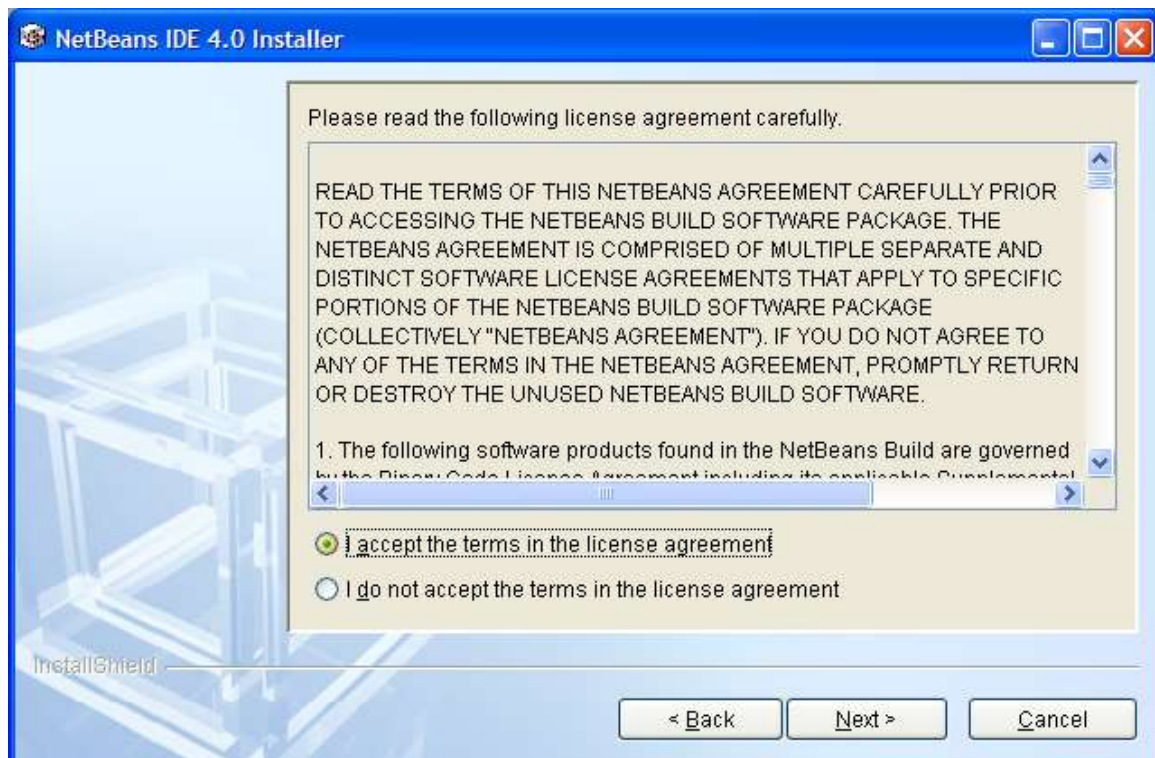


Step 2: Run the installer

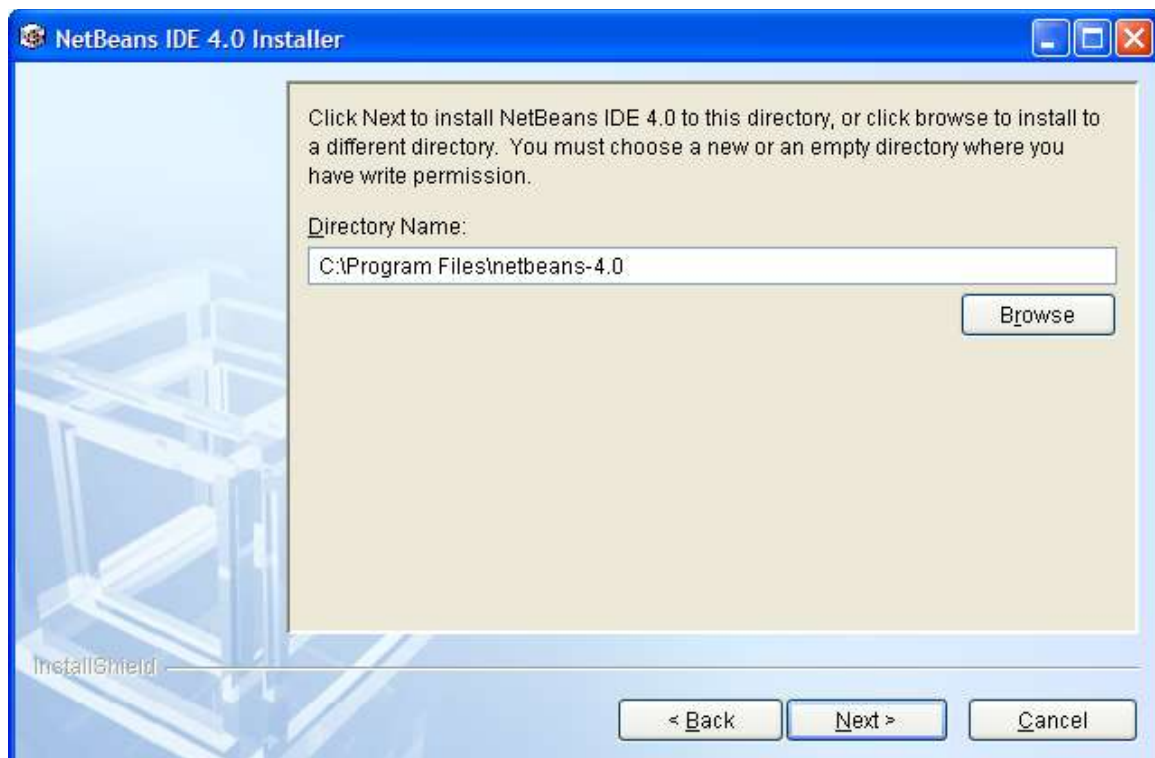
To run the installer, just double-click on the installer icon. After clicking on the netbeans-4_0-bin-windows icon, the Netbeans installation wizard will appear. Click on NEXT to enter installation process.



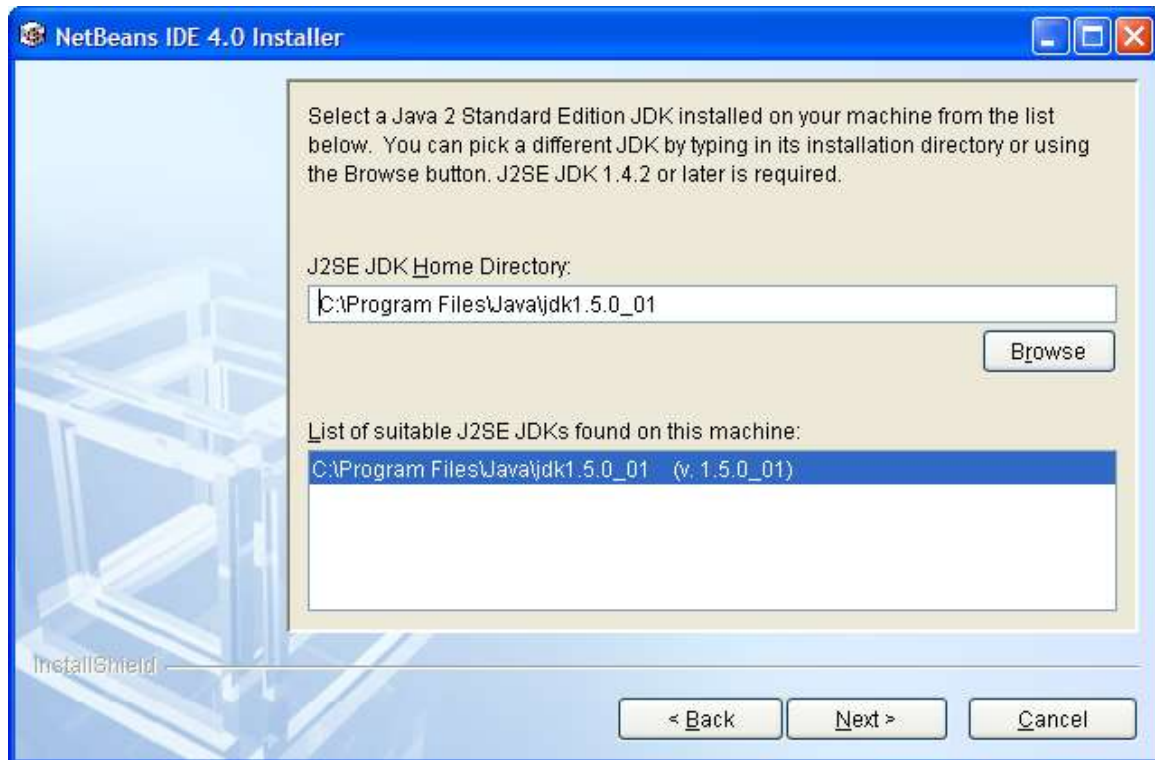
The agreement page will the appear. Choose to ACCEPT and click NEXT to continue.



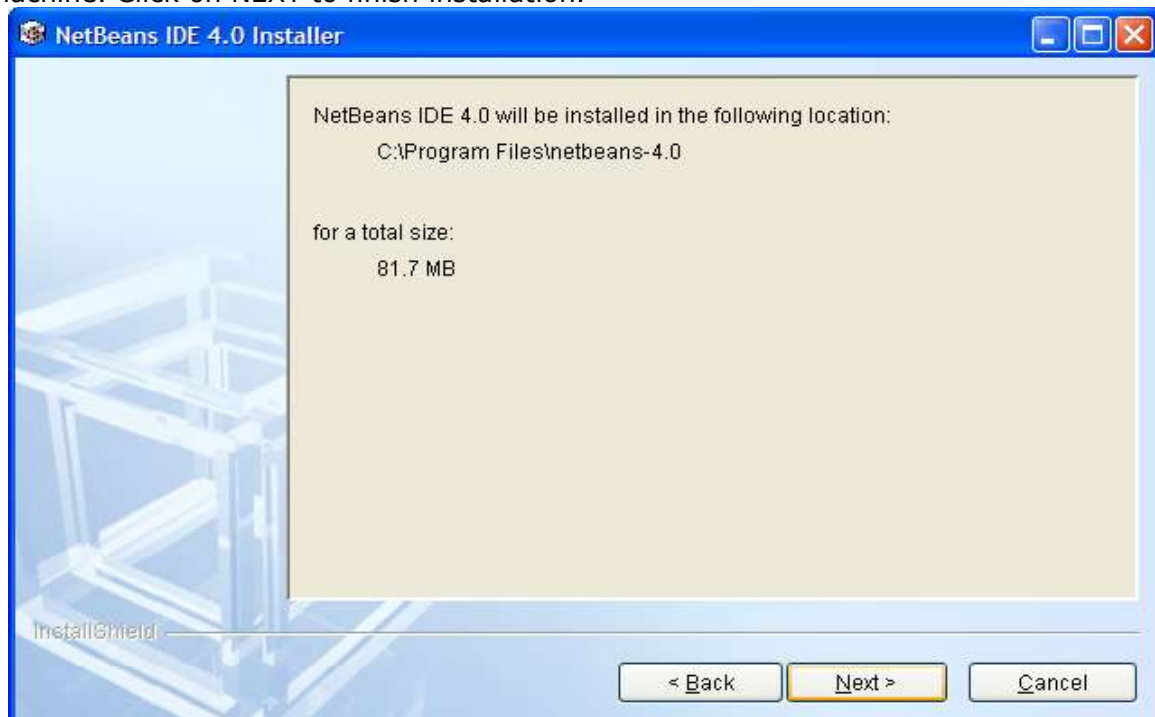
Then you will be given the choice on which directory to place the Netbeans. You can move on by clicking NEXT or you can click on BROWSE to choose a different directory.



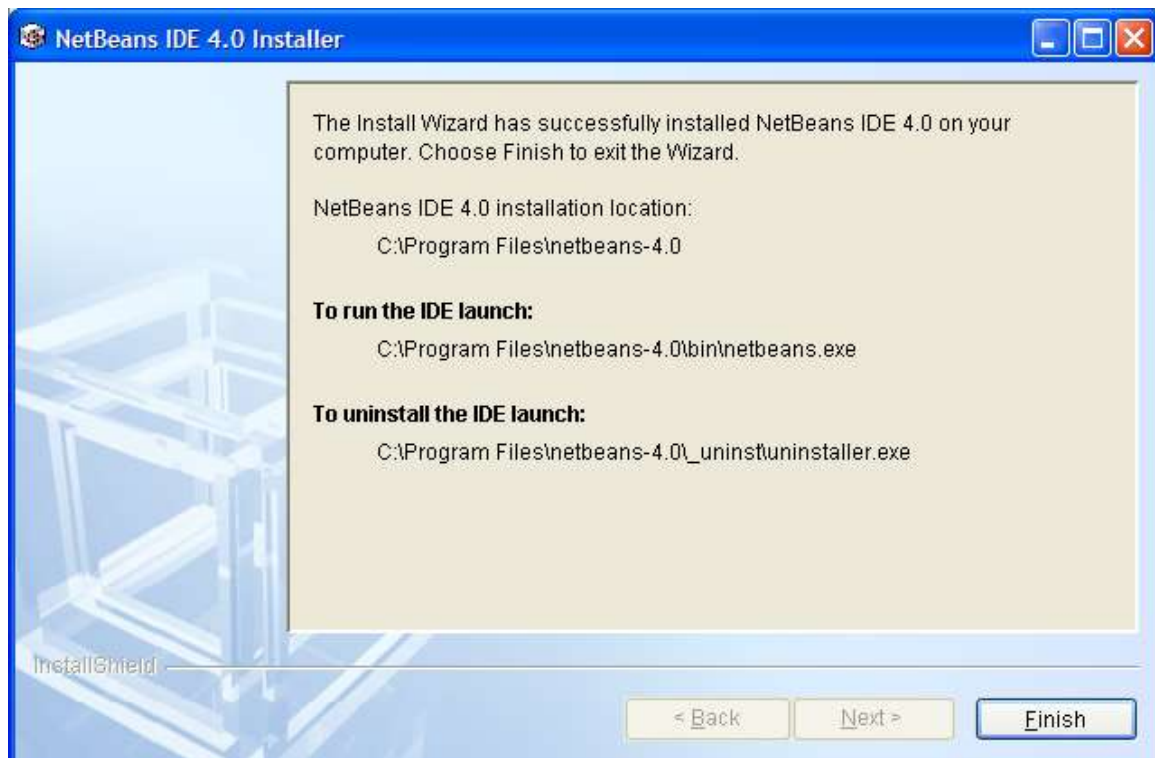
Next is choosing the Standard Edition JDKs from your machine. If you have finished installing Java, the `jdk1.5.0_01` should appear from your choices. Click on NEXT to continue.



It will then inform you the location and size of Netbeans which will be installed to your machine. Click on NEXT to finish installation.



You have installed Netbeans on your computer. Click on FINISH to complete installation.



Appendix B: Getting to know your Programming Environment (Windows XP version)

In this section, we will be discussing on how to write, compile and run Java programs. There are two ways of doing this, the first one is by using a console and a text editor. The second one is by using Netbeans which is an **Integrated Development Environment or IDE**.

An IDE is a programming environment integrated into a software application that provides a GUI builder, a text or code editor, a compiler and/or interpreter and a debugger.

Before going into details, let us first take a look at the first Java program you will be writing.

My First Java Program

```
public class Hello
{
    /**
     * My first java program
     */
    public static void main(String[] args) {

        //prints the string "Hello world" on screen
        System.out.println("Hello world!");

    }
}
```

Before we try to explain what the program means, let's first try to write this program in a file and try to run it.

Using a Text Editor and Console

For this example, we will be using the text editor "Notepad"(for Windows) to edit the Java source code. You can use other text editors if you want to. You will also need to open the MS-DOS prompt window to compile and execute your Java programs.

Step 1: Start Notepad

To start Notepad in Windows, click on start-> All Programs-> Accessories-> Notepad.

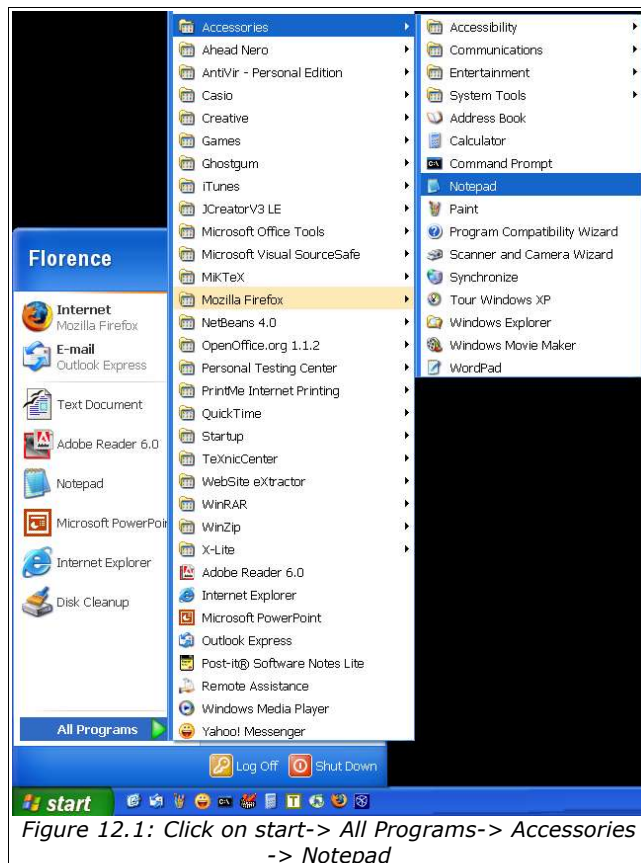


Figure 12.1: Click on start-> All Programs-> Accessories
-> Notepad

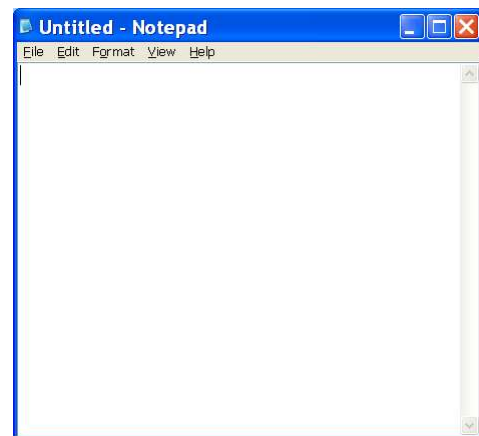


Figure 12.2: Notepad Application

Step 2: Open the Command Prompt window

To open the MSDOS command prompt in Windows, click on start-> All programs-> Accessories-> Command Prompt.

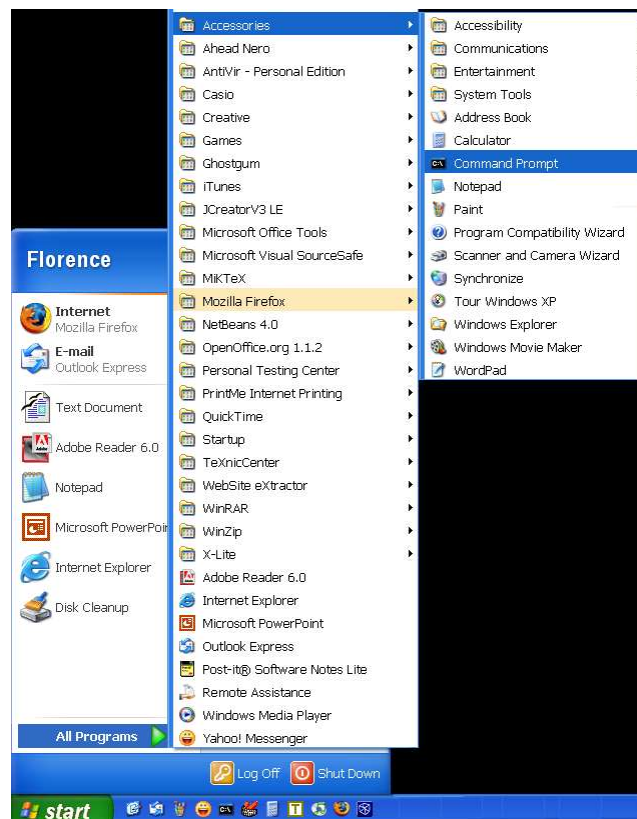


Figure 12.3: start-> All programs-> Accessories -> Command Prompt

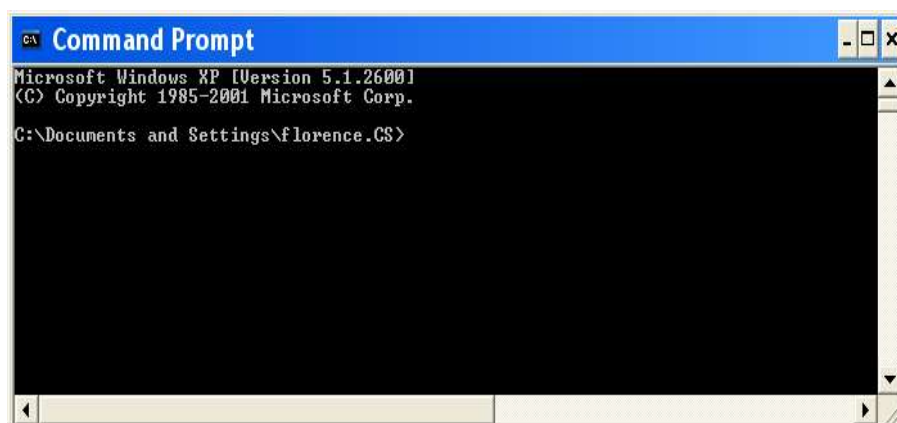
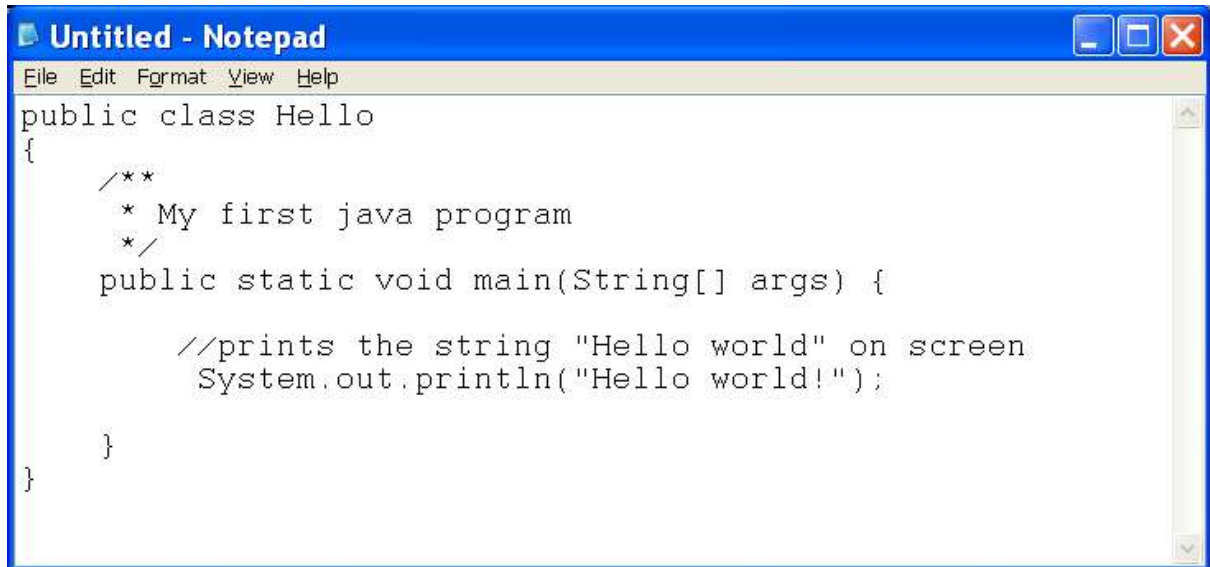


Figure 12.4: MSDOS Command Prompt

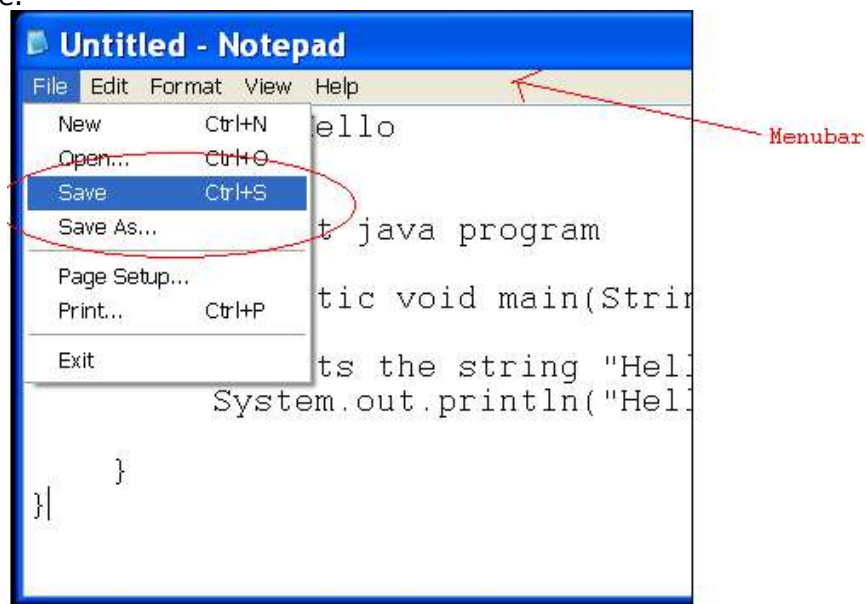
Step 3: Write your the source code of your Java program in Notepad

```
public class Hello
{
    /**
     * My first java program
     */
    public static void main(String[] args) {
        //prints the string "Hello world" on screen
        System.out.println("Hello world!");
    }
}
```

Step 4: Save your Java Program

We will save our program on a file named "Hello.java", and we will be saving it inside a folder named MYJAVAPROGRAMS.

To open the **Save** dialog box, click on the File menu found on the menubar and then click on Save.



After doing the procedure described above, a dialog box will appear as shown in Figure below.

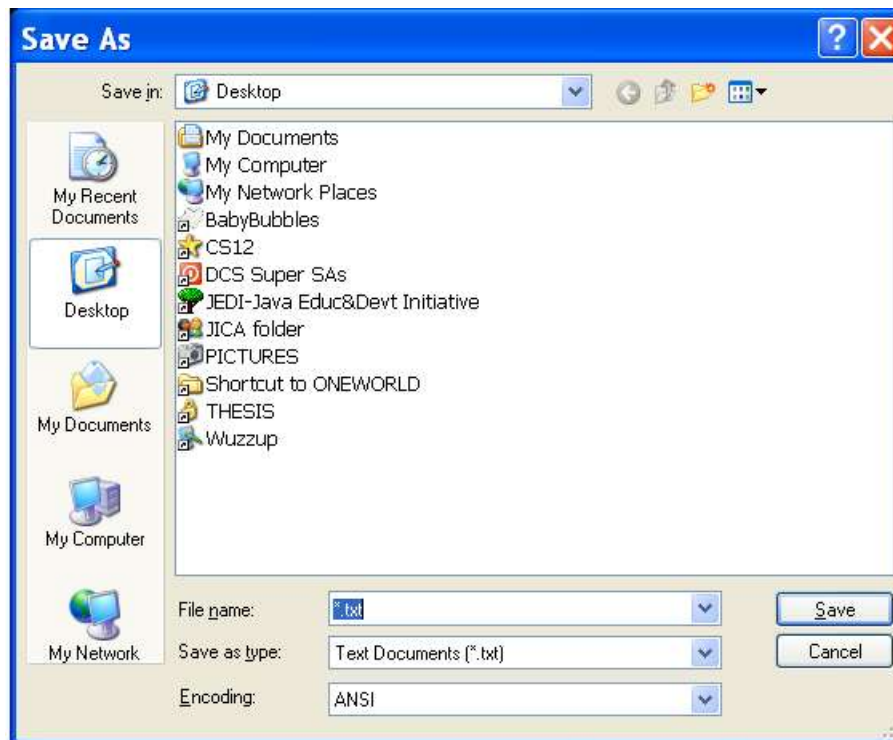


Figure 12.5: This Dialog appears after clicking on File -> Save

Click on the **MY DOCUMENTS** button to open the My Documents folder where we will be saving all your Java programs.

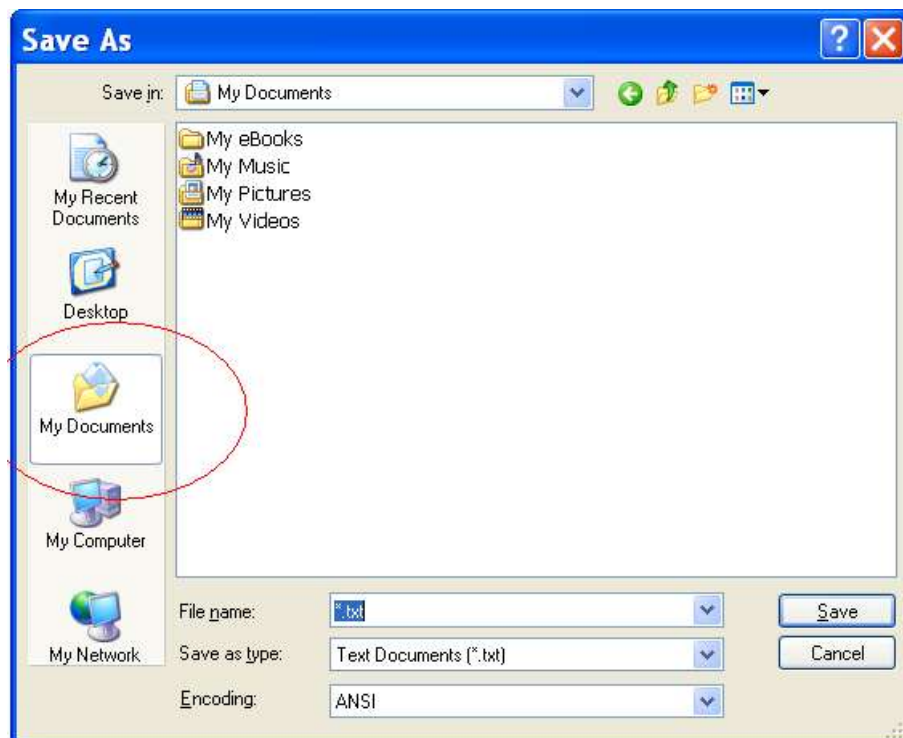


Figure 12.6: Click on the button encircled. This will open your "My Documents" folder

Now, we'll create a new folder inside the My Documents folder where we will save your programs. We shall name this folder MYJAVAPROGRAMS. Click on the button encircled in the figure below to create the folder.

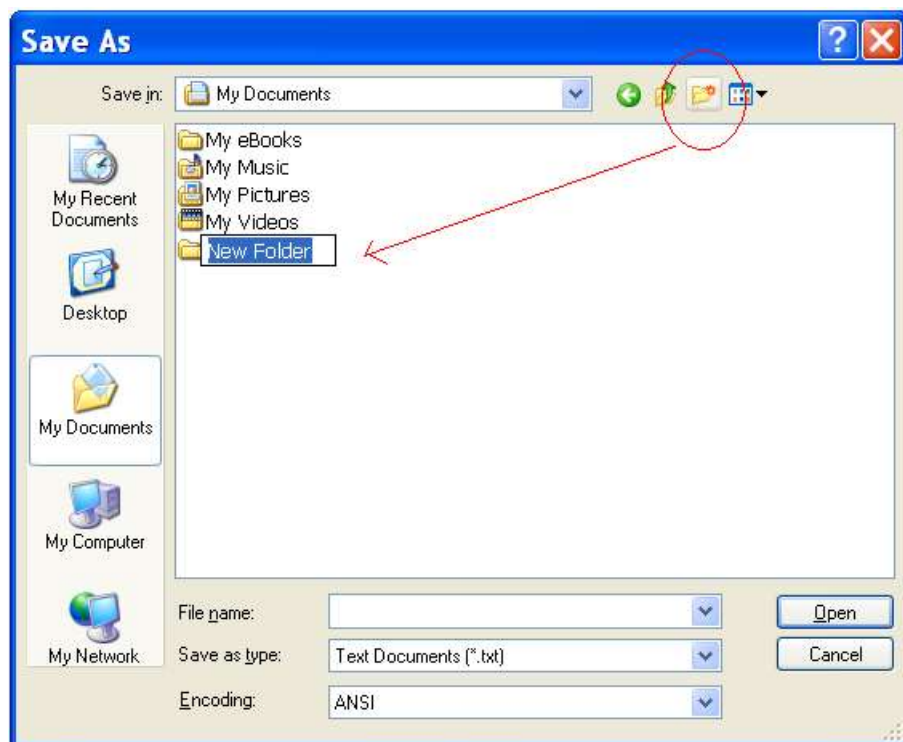
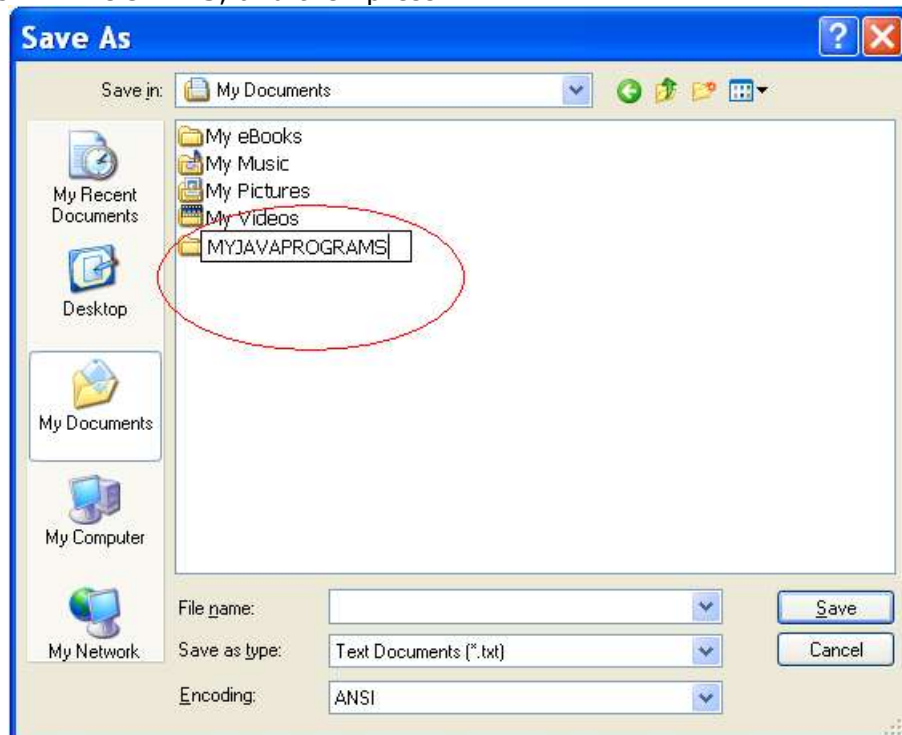
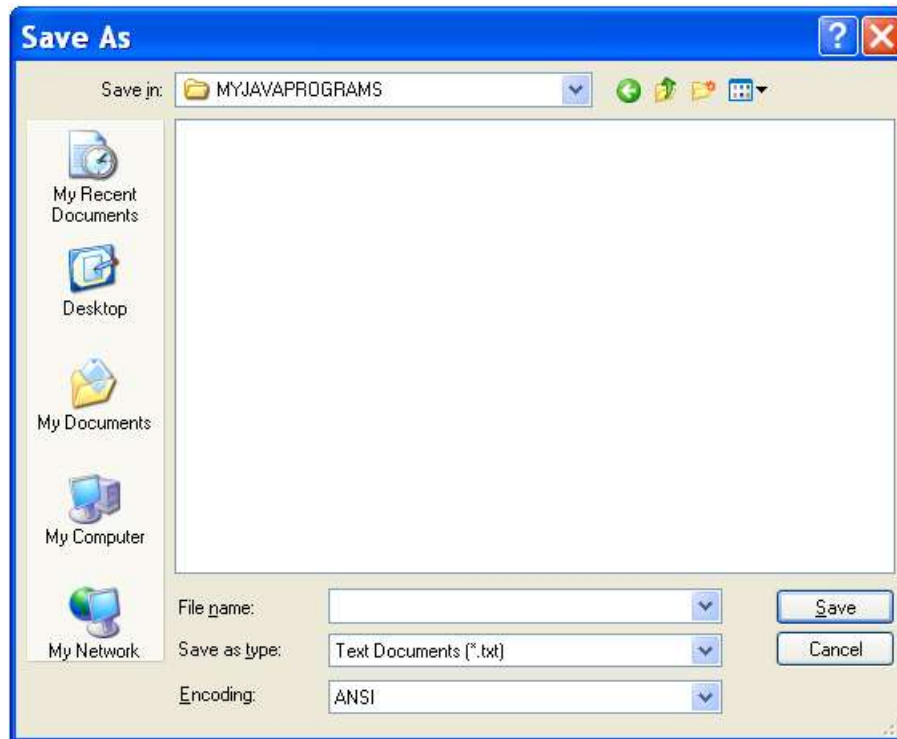


Figure 12.7: Clicking on the encircled button will create a New Folder.

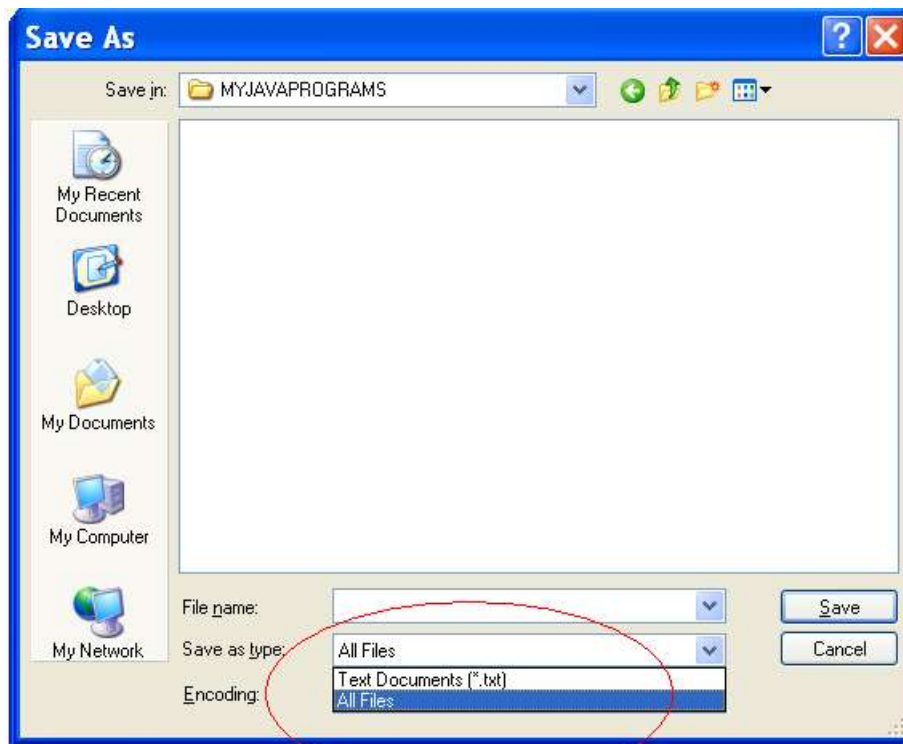
After the folder is created, you can type in the desired name for this folder. In this case, type in MYJAVAPROGRAMS, and then press ENTER.



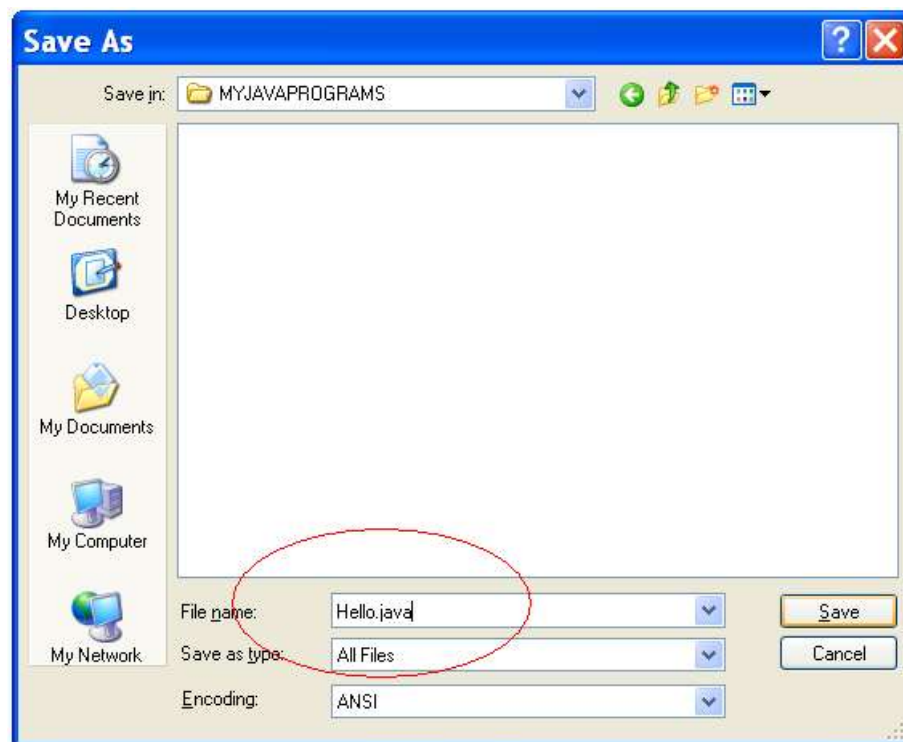
Now that we've created the folder where we will save all the files, double click on that folder to open it. You will see a similar figure as shown below. The folder should be empty for now since it's a newly created folder and we haven't saved anything in it yet.



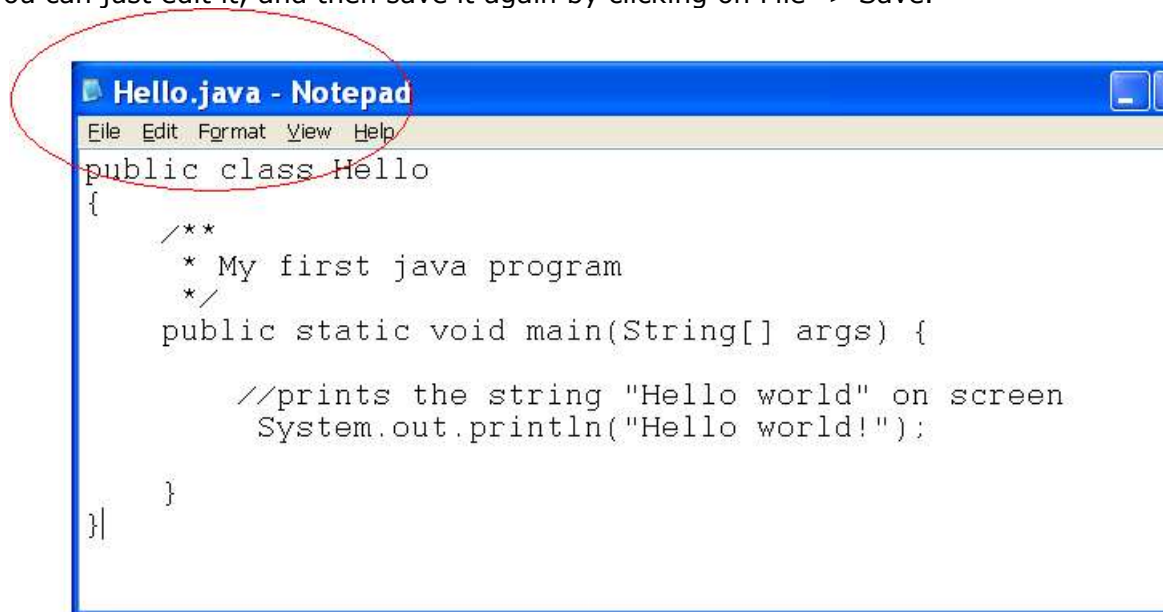
Now click on the drop down list box "Save as type", so that we can choose what kind of file we want to save. Click on the "All Files" option.



Now, in the Filename textbox, type in the filename of your program, which is "Hello.java", and then click on the SAVE button.



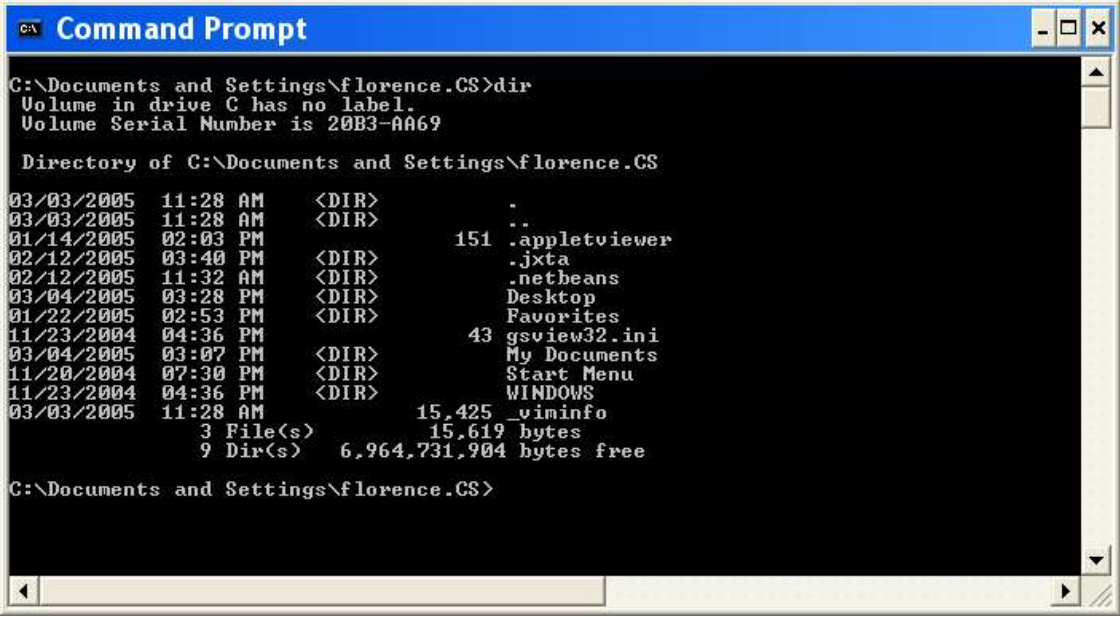
Now that you've saved your file, notice how the title of the frame changes from Untitled-Notepad to Hello.java-Notepad. Take note that if you want to make changes in your file, you can just edit it, and then save it again by clicking on File -> Save.



Step 5: Compiling your program

Now, the next step is to compile your program. Go to the MSDOS command prompt window we just opened a while ago.

Typically, when you open the command prompt window, it opens up and takes you directly to what is called your **home folder**. To see what is inside that home folder, type **DIR** or **dir** and then press ENTER. What you will see is a list of files and folders inside your home folder.



```
C:\Documents and Settings\florence.CS>dir
Volume in drive C has no label.
Volume Serial Number is 20B3-AA69

Directory of C:\Documents and Settings\florence.CS

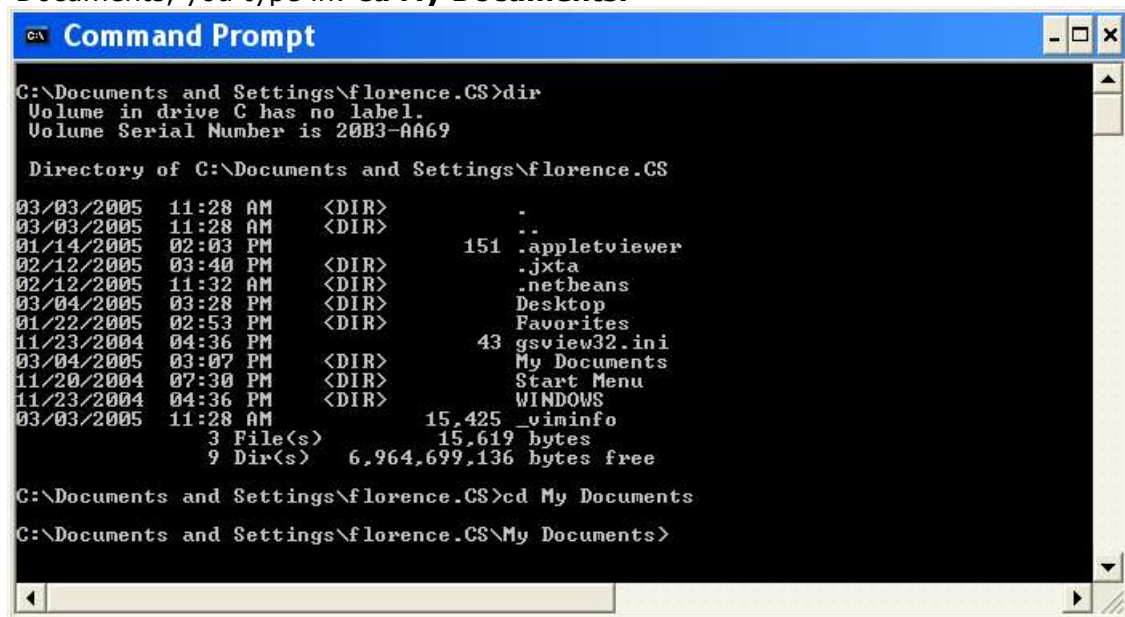
03/03/2005  11:28 AM  <DIR>          .
03/03/2005  11:28 AM  <DIR>          ..
01/14/2005  02:03 PM             151 .appletviewer
02/12/2005  03:40 PM  <DIR>          .jxta
02/12/2005  11:32 AM  <DIR>          .netbeans
03/04/2005  03:28 PM  <DIR>          Desktop
01/22/2005  02:53 PM  <DIR>          Favorites
11/23/2004  04:36 PM             43 gsview32.ini
03/04/2005  03:07 PM  <DIR>          My Documents
11/20/2004  07:30 PM  <DIR>          Start Menu
11/23/2004  04:36 PM  <DIR>          WINDOWS
03/03/2005  11:28 AM             15,425 _vininfo
               3 File(s)              15,619 bytes
               9 Dir(s)  6,964,731,904 bytes free

C:\Documents and Settings\florence.CS>
```

Figure 12.8: List of files and folders shown after executing the command DIR.

Now, you can see here that there is a folder named "My Documents" where we created your MYJAVAPROGRAMS folder. Now let's go inside that directory.

To go inside a directory, you type in the command: **cd [directory name]**. The "cd" command stands for, change directory. In this case, since the name of our directory is My Documents, you type in: **cd My Documents**.



```

C:\Documents and Settings\florence.CS>dir
Volume in drive C has no label.
Volume Serial Number is 20B3-AA69

Directory of C:\Documents and Settings\florence.CS

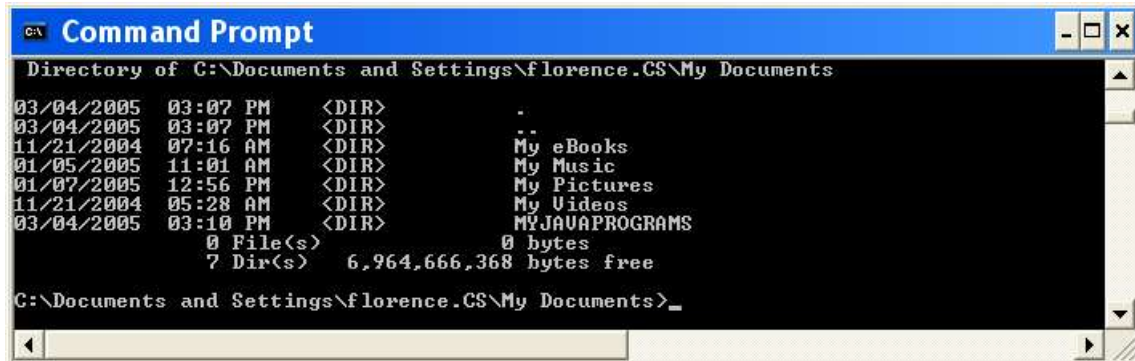
03/03/2005  11:28 AM    <DIR>        .
03/03/2005  11:28 AM    <DIR>        ..
01/14/2005  02:03 PM             151  .appletviewer
02/12/2005  03:40 PM    <DIR>        .jxta
02/12/2005  11:32 AM    <DIR>        .netbeans
03/04/2005  03:28 PM    <DIR>        Desktop
01/22/2005  02:53 PM    <DIR>        Favorites
11/23/2004  04:36 PM             43  gsview32.ini
03/04/2005  03:07 PM    <DIR>        My Documents
11/20/2004  07:30 PM    <DIR>        Start Menu
11/23/2004  04:36 PM    <DIR>        WINDOWS
03/03/2005  11:28 AM             15,425  _viminfo
               3 File(s)              15,619 bytes
               9 Dir(s)  6,964,699,136 bytes free

C:\Documents and Settings\florence.CS>cd My Documents
C:\Documents and Settings\florence.CS\My Documents>

```

Figure 12.9: Inside the My Documents folder

Now that you are inside the "My Documents" folder, try typing in the "dir" command again, and tell me what you see.



```

C:\Documents and Settings\florence.CS\My Documents>dir
Directory of C:\Documents and Settings\florence.CS\My Documents

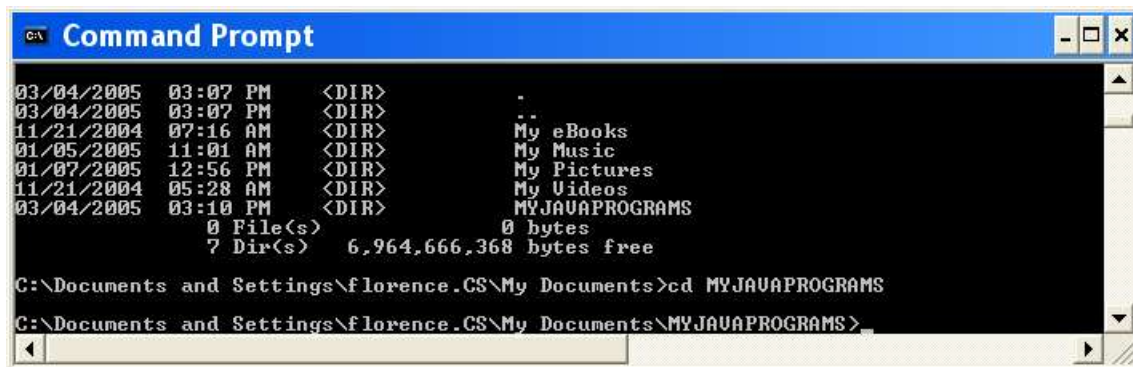
03/04/2005  03:07 PM    <DIR>        .
03/04/2005  03:07 PM    <DIR>        ..
11/21/2004  07:16 AM    <DIR>        My eBooks
01/05/2005  11:01 AM    <DIR>        My Music
01/07/2005  12:56 PM    <DIR>        My Pictures
11/21/2004  05:28 AM    <DIR>        My Videos
03/04/2005  03:10 PM    <DIR>        MYJAVAPROGRAMS
               0 File(s)              0 bytes
               7 Dir(s)  6,964,666,368 bytes free

C:\Documents and Settings\florence.CS\My Documents>_

```

Figure 12.10: The contents of My Documents

Now perform the same steps described before to go inside the MYJAVAPROGRAMS folder.



```
C:\ Command Prompt

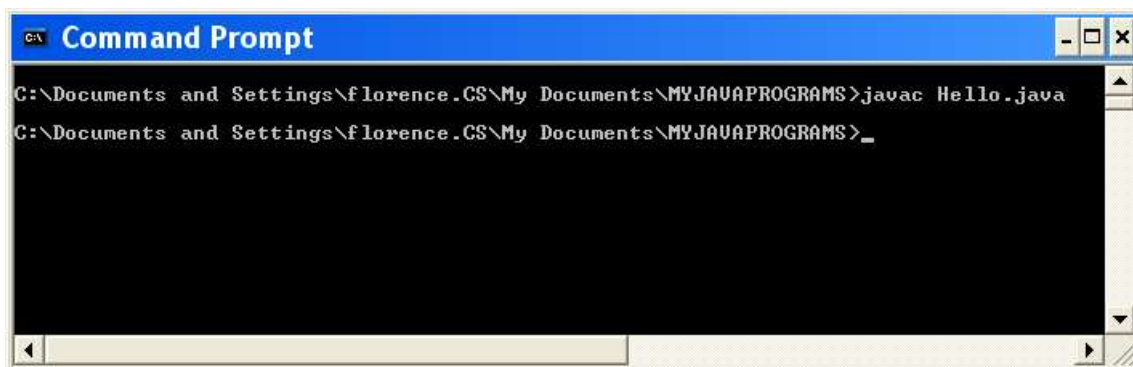
03/04/2005  03:07 PM  <DIR>          .
03/04/2005  03:07 PM  <DIR>          ..
11/21/2004  07:16 AM  <DIR>          My eBooks
01/05/2005  11:01 AM  <DIR>          My Music
01/07/2005  12:56 PM  <DIR>          My Pictures
11/21/2004  05:28 AM  <DIR>          My Videos
03/04/2005  03:10 PM  <DIR>          MYJAVAPROGRAMS
               0 File(s)              0 bytes
               7 Dir(s)  6,964,666,368 bytes free

C:\Documents and Settings\florence.CS\My Documents>cd MYJAVAPROGRAMS
C:\Documents and Settings\florence.CS\My Documents\MYJAVAPROGRAMS>
```

Figure 12.11: Inside the MYJAVAPROGRAMS folder

Once inside the folder where your Java programs are, let us now start compiling your Java program. Take note that, you should make sure that the file is inside the folder where you are in. In order to do that, execute the **dir** command again to see if your file is inside that folder.

To compile a Java program, we type in the command: **javac [filename]**. So in this case, type in: **javac Hello.java**.



```
C:\ Command Prompt

C:\Documents and Settings\florence.CS\My Documents\MYJAVAPROGRAMS>javac Hello.java
C:\Documents and Settings\florence.CS\My Documents\MYJAVAPROGRAMS>
```

Figure 12.12: Compile program by using the **javac** command

During compilation, javac adds a file to the disk called **[filename].class**, or in this case, **Hello.class**, which is the actual bytecode.

Step 6: Running the Program

Now, assuming that there are no problems during compilation (we'll explore more of the problems encountered during compilation in the next section), we are now ready to run your program.

To run your Java program, type in the command: **java [filename without the extension]**, so in the case of our example, type in: **java Hello**

You can see on the screen that you have just run your first Java program that prints the message, "Hello world!".

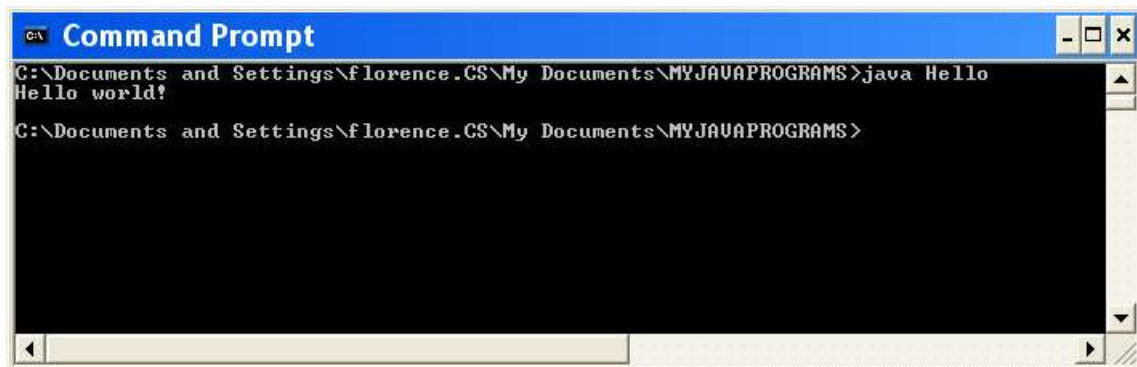


Figure 12.13: Output of the program

Setting the Path

Sometimes, when you try to invoke the `javac` or `java` command, you encounter the message: **'javac' is not recognized as an internal or external command, operable program or batch file.** This means that either you haven't installed Java in your system yet, or you have to configure the path on where the Java commands are installed so that your system will know where to find them.

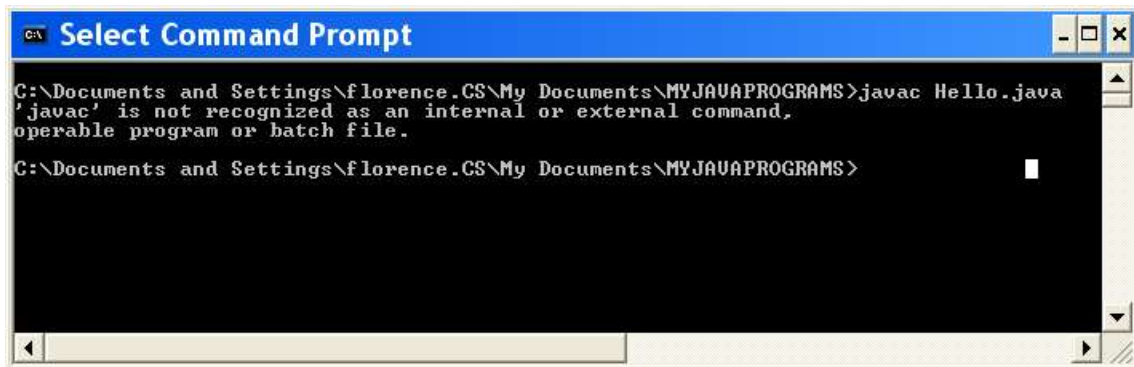


Figure 12.14: System did not recognize the `javac` command

If you are sure that you've already installed Java in your system, try setting the `PATH` variable to point to where the Java commands are installed. To do this, type in the command: **`set PATH=C:\j2sdk1.4.2_04\bin`**. This will tell your system to look for the commands in the **`C:\j2sdk1.4.2_04\bin`** folder, which is usually the default location wherein your Java files are placed during installation. After doing this, you can now use the Java commands.

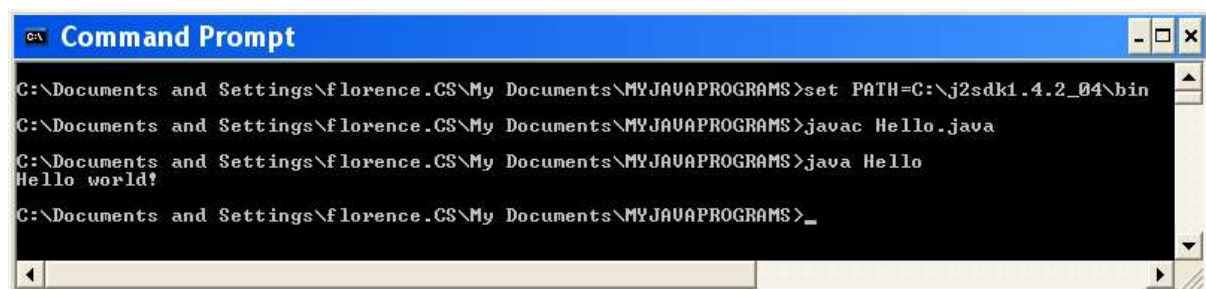


Figure 12.15: Setting the path and running `java`

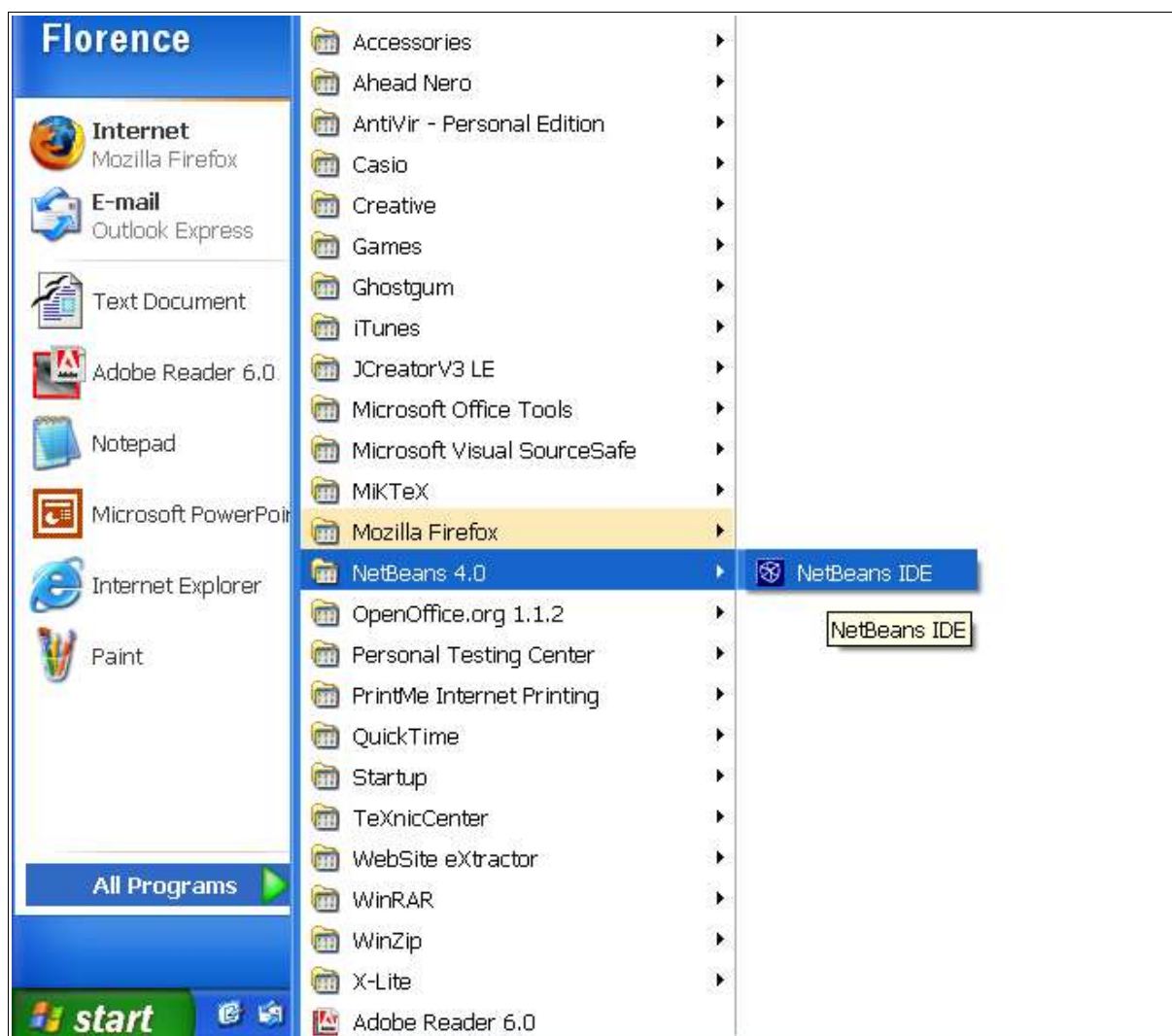
Using Netbeans

Now that we've tried doing our programs the complicated way, let's now see how to do all the processes we've described in the previous sections by using just one application.

In this part of the lesson, we will be using **Netbeans**, which is an **Integrated Development Environment or IDE**. An IDE is a programming environment integrated into a software application that provides a GUI builder, a text or code editor, a compiler and/or interpreter and a debugger.

Step 1: Run Netbeans

To run Netbeans, click on start-> All Programs-> NetBeans 4.0 -> NetBeans IDE



After you've open NetBeans IDE, you will see a graphical user interface (GUI) similar to what is shown below.

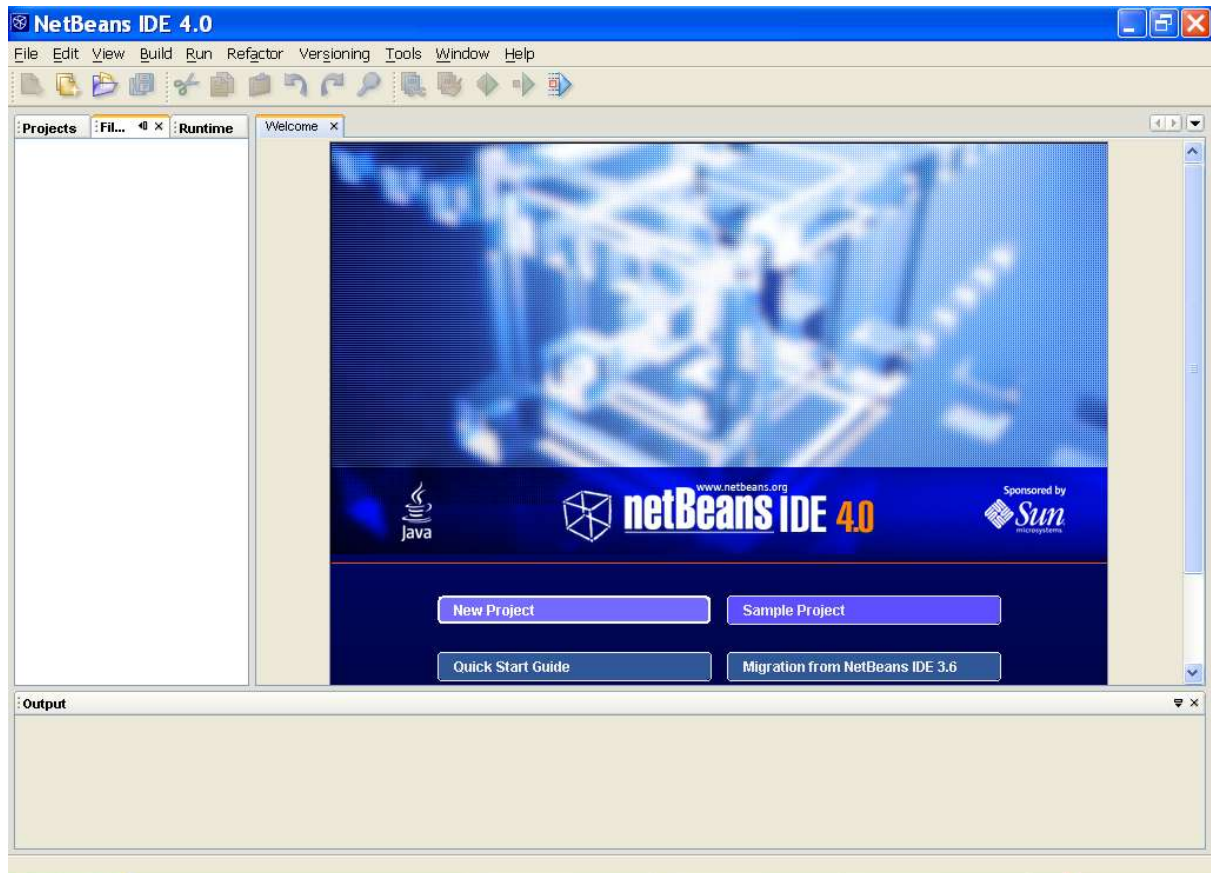
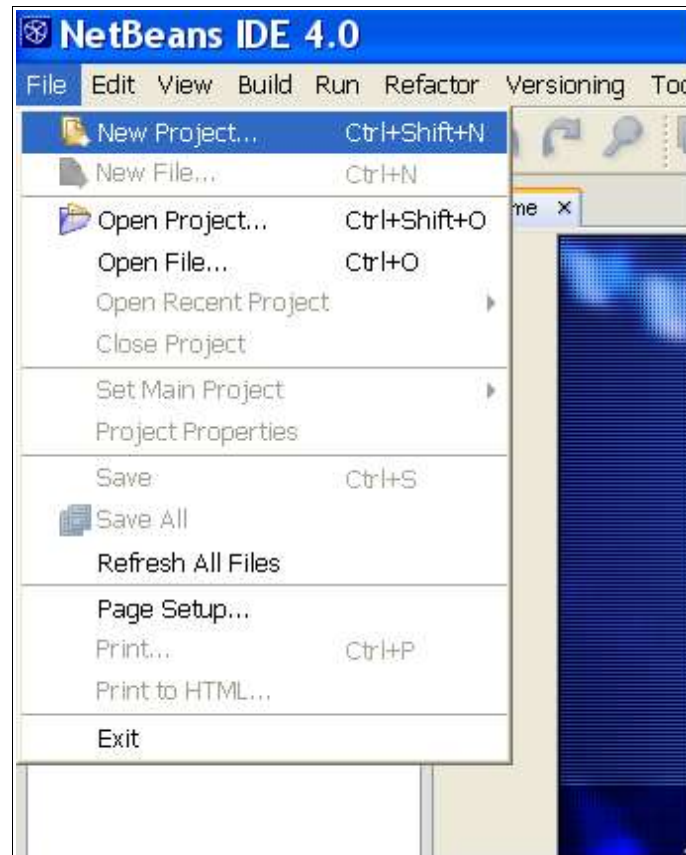


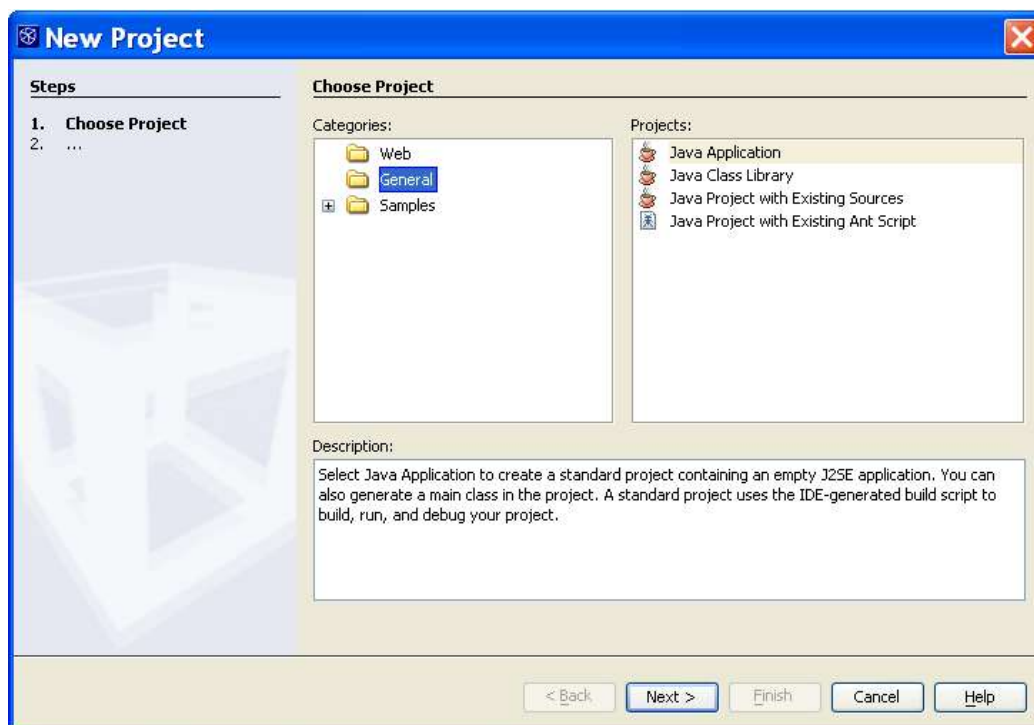
Figure 12.16: NetBeans IDE

Step 2: Make a project

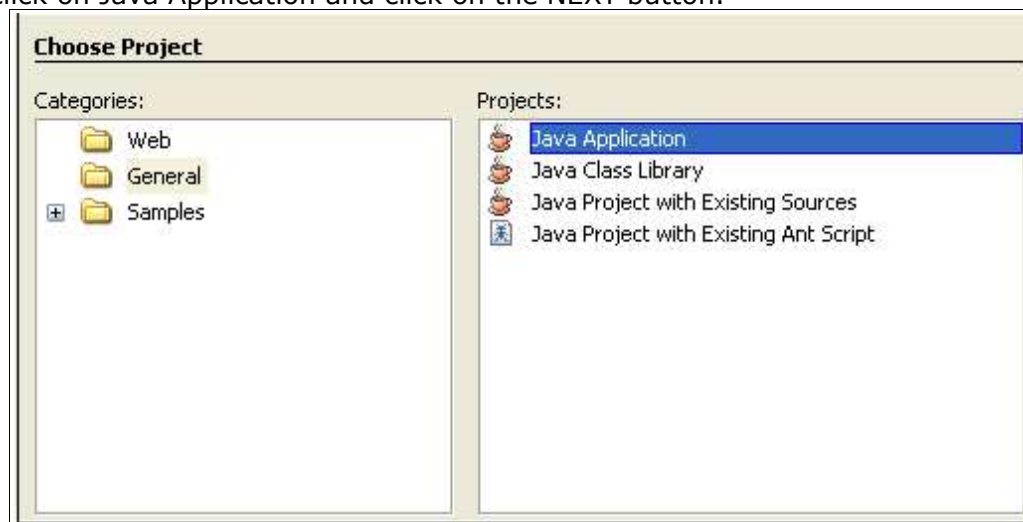
Now, let's first make a project. Click on File-> New Project.



After doing this, a New Project dialog will appear.



Now click on Java Application and click on the NEXT button.



Now, a New Application dialog will appear. Edit the Project Name part and type in "HelloApplication".

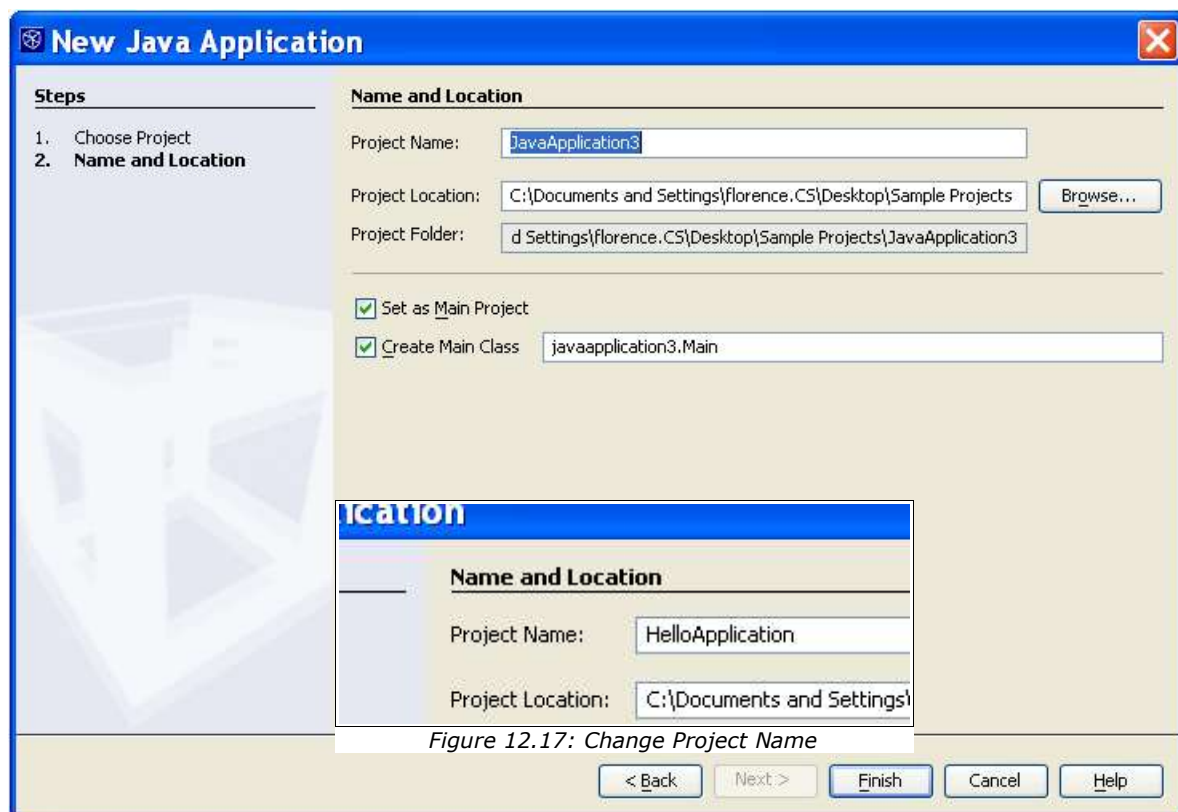
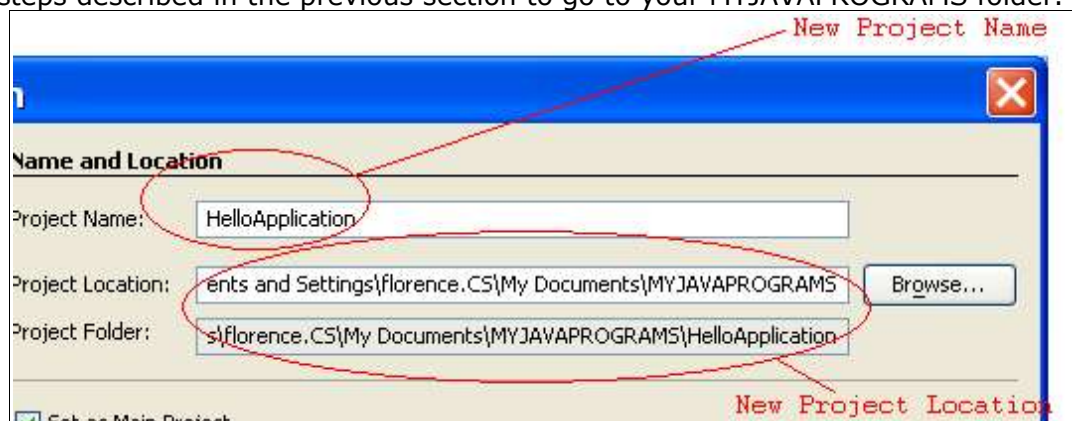
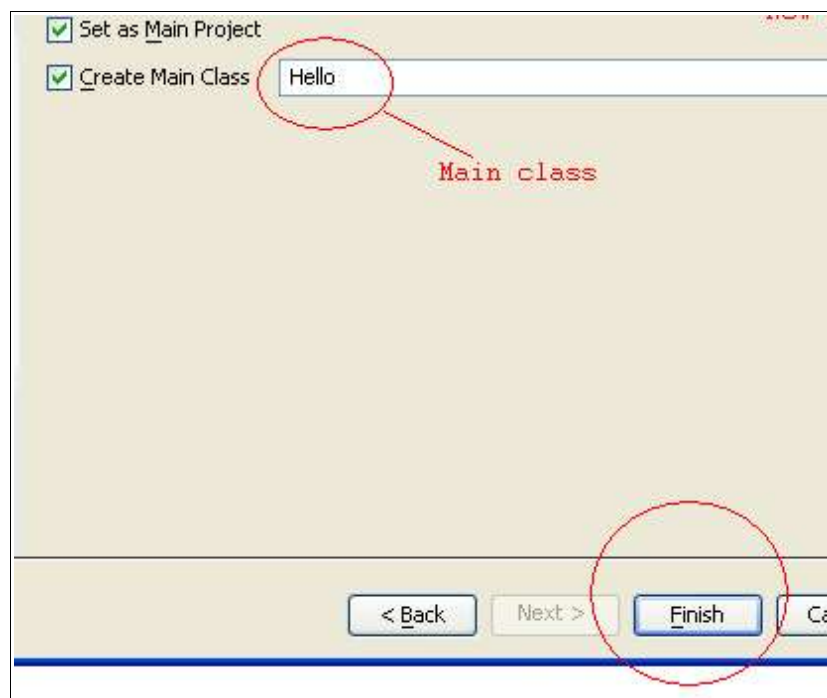


Figure 12.17: Change Project Name

Now try to change the Application Location, by clicking on the BROWSE button. Follow the steps described in the previous section to go to your MYJAVAPROGRAMS folder.



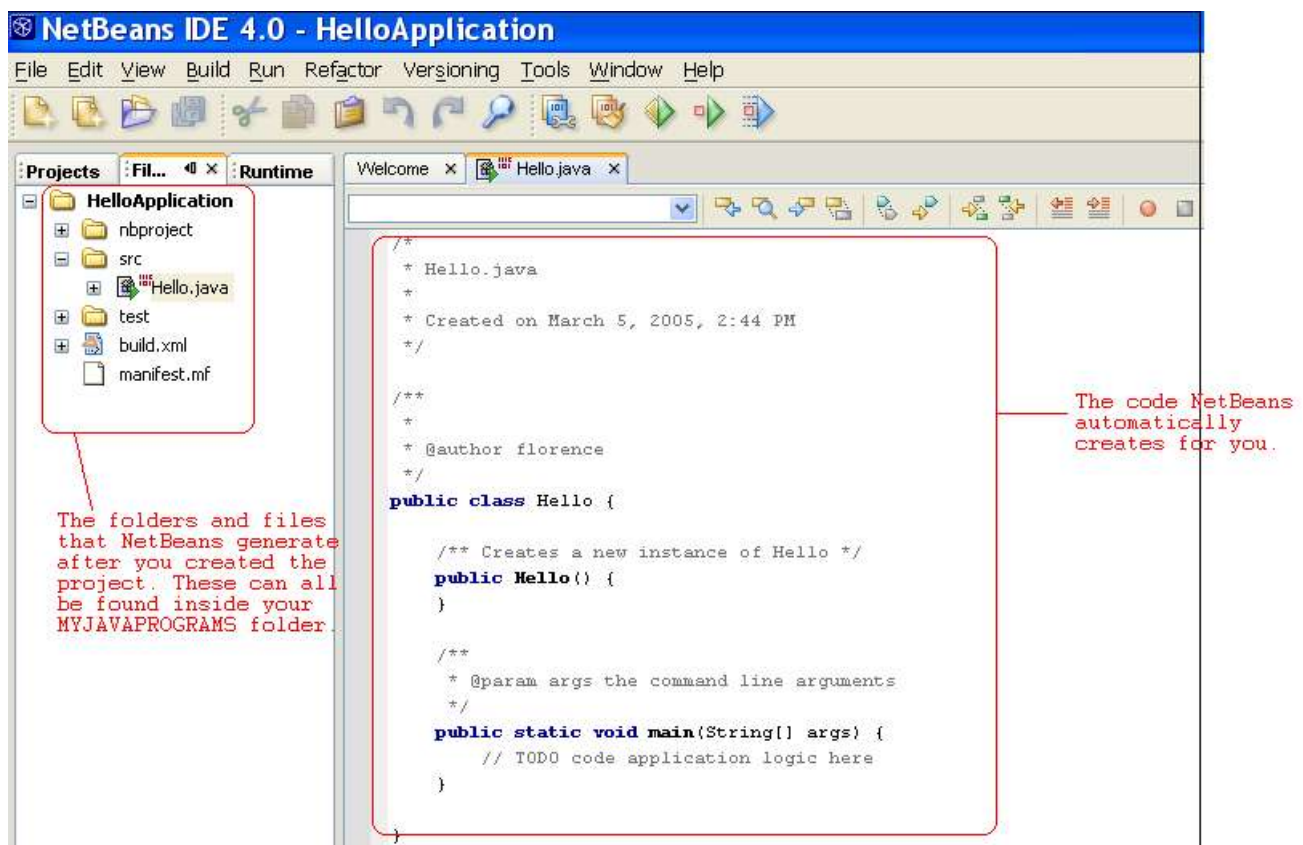
Finally, on the Create Main Class textfield, type in Hello as the main class' name, and then click on the FINISH button.



Step 3: Type in your program

Before typing in your program, let us first describe the main window after creating the project.

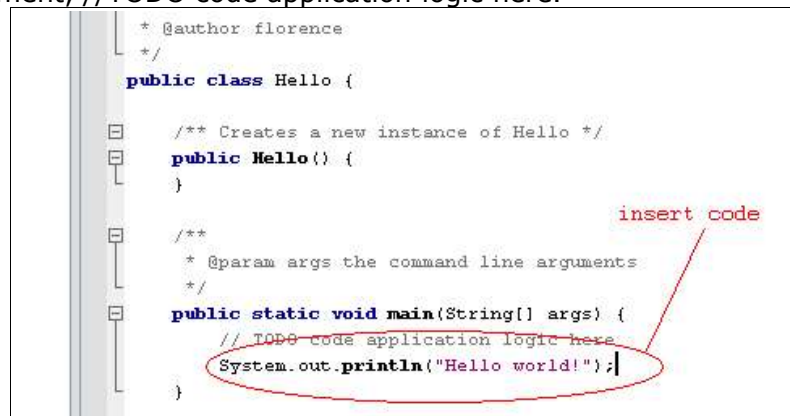
As shown below, NetBeans automatically creates the basic code for your Java program. You can just add your own statements to the generated code. On the left side of the window, you can see a list of folders and files that NetBeans generated after creating the project. This can all be found in your MYJAVAPROGRAMS folder, where you set the Project location.



Now, try to modify the code generated by Netbeans. Ignore the other parts of the program for now, as we will explain the details of the code later. Insert the code:

```
System.out.println("Hello world!");
```

after the statement, `//TODO code application logic here`.



Step 4: Compile your program

Now, to compile your program, just click on Build -> Build Main Project. Or, you could also use the shortcut button to compile your code.

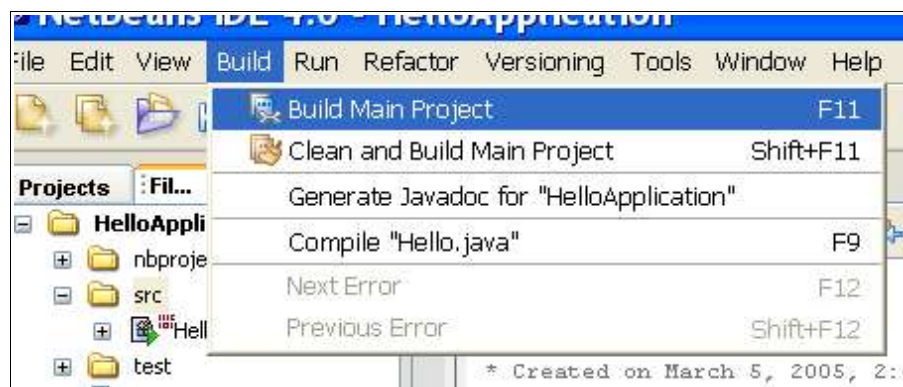


Figure 12.18: Shortcut button to compile code

If there are no errors in your program, you will see a build successful message on the output window.

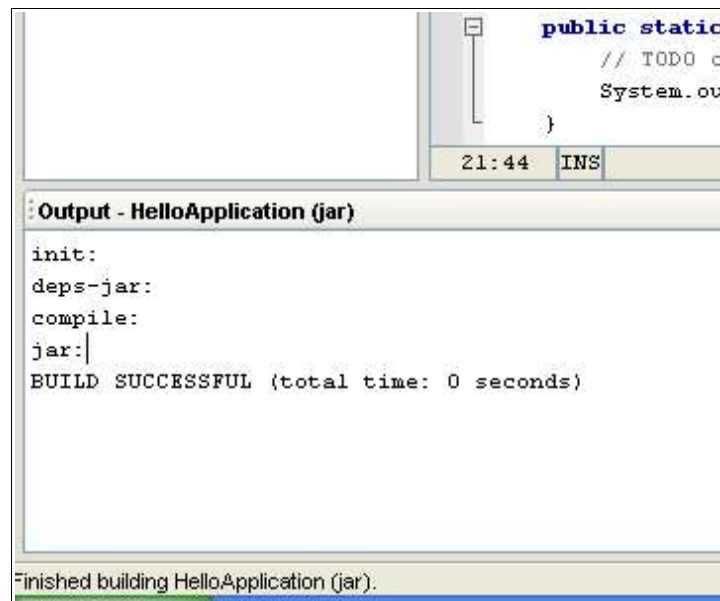


Figure 12.19: Output window just below the window where you type your source code

Step 5: Run your program

To run your program, click on Run-> Run Main Project. Or you could also use the shortcut button to run your program.

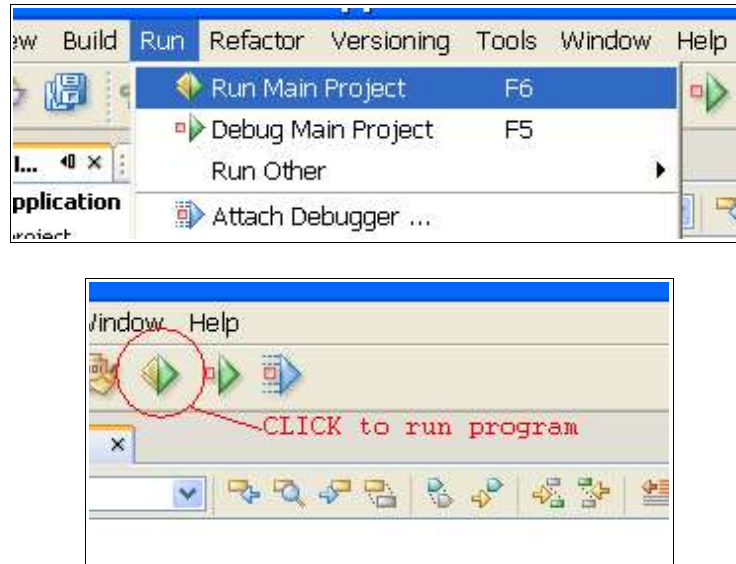


Figure 12.20: Shortcut button to run program

The output of your program is displayed in the output window.

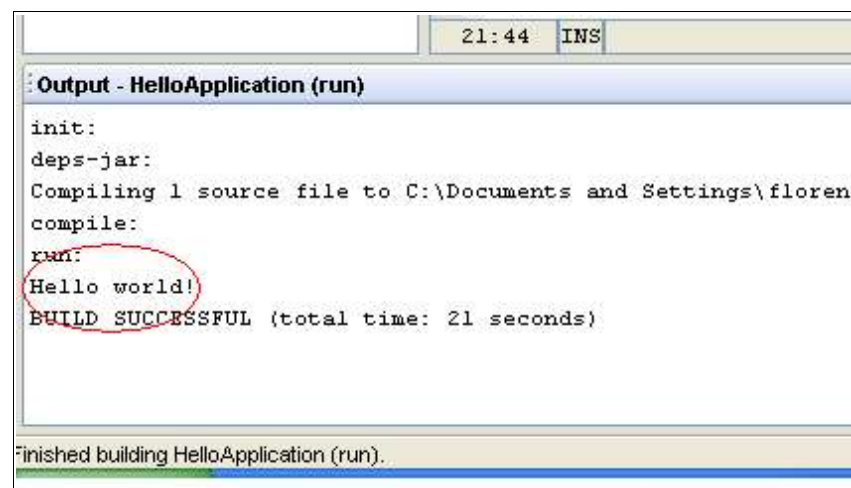


Figure 12.21: Output of Hello.java

Appendix C : Answers to Exercises

Chapter 1 Exercises

1.1 Writing Algorithms

1. Baking Bread

Pseudocode:

```
prepare all ingredients
pour all ingredients in mixing bowl

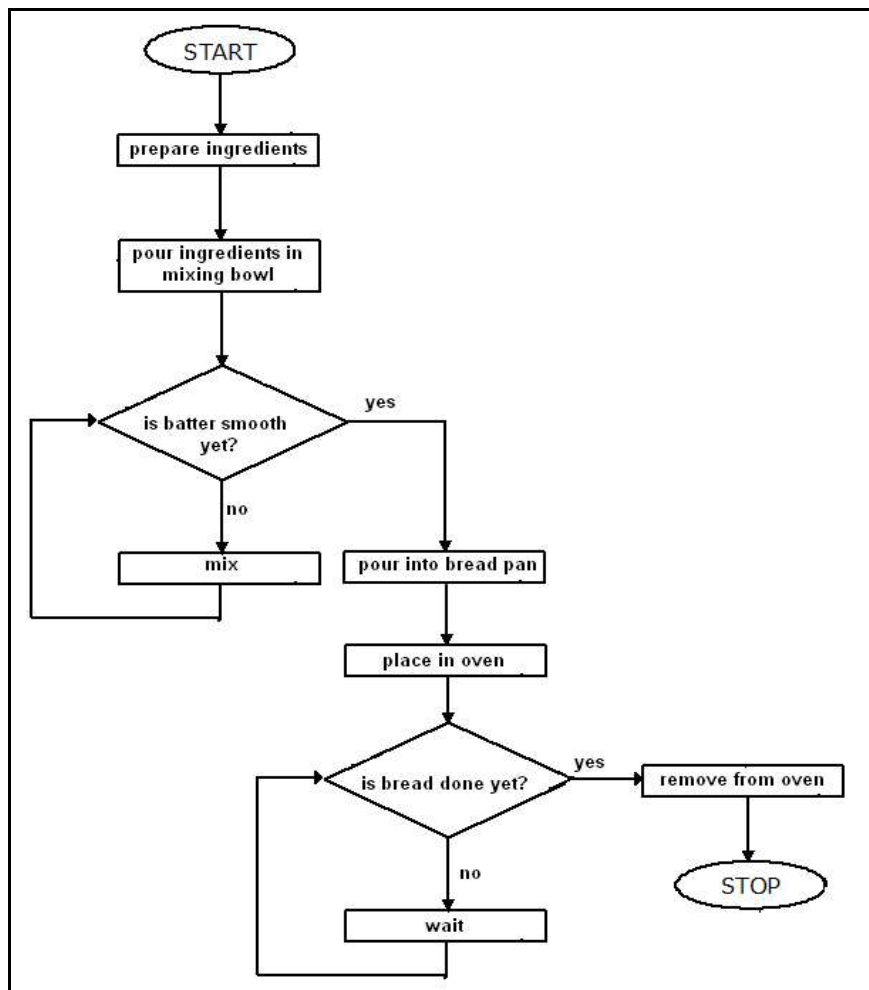
while batter not smooth yet
    mix ingredients

pour into bread pan
place inside oven

while bread not yet done
    wait

remove from oven
```

Flowchart:



2. Logging into your laboratory's computer

Pseudocode:

```
Let power = computer's power button
Let in = status of user (initially false)

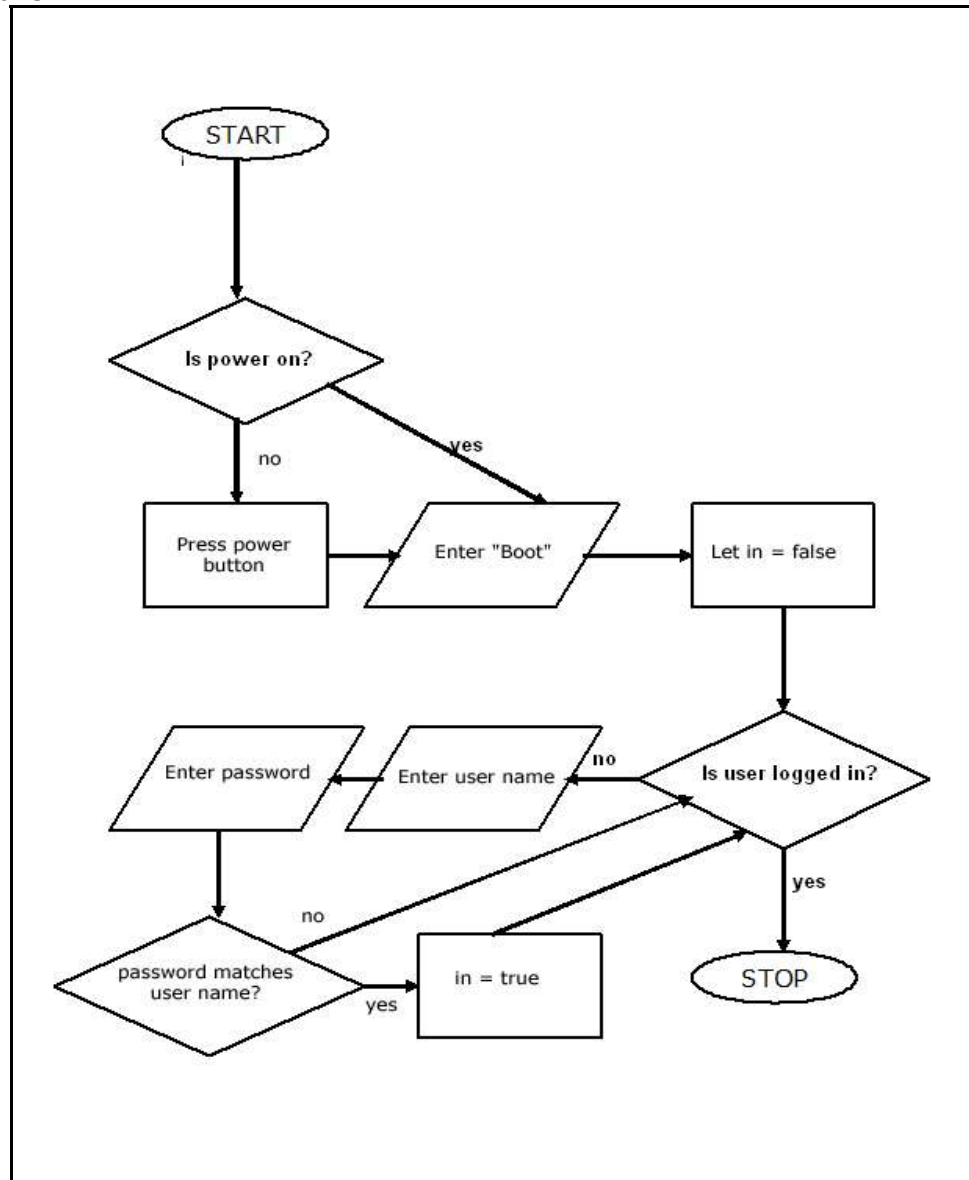
if power == off
    Press power button

Enter "boot" process

while in == false
    enter user name
    enter password

    if password and user name correct
        in = true
end while
```

Flowchart:



3. Getting the average of three numbers

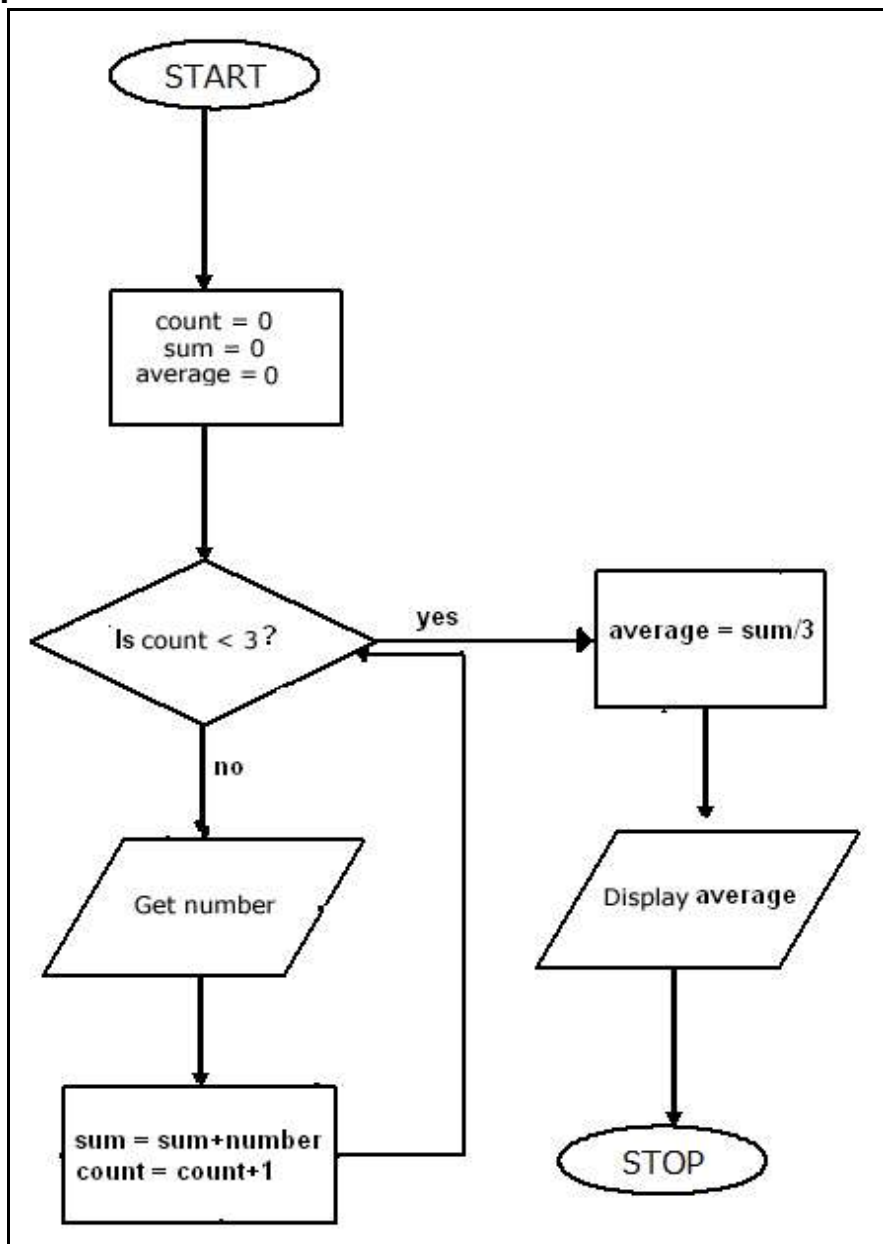
Pseudocode:

```
Let count = 0
Let sum = 0
Let average = 0

While count < 3
  Get number
  sum = sum + number
  count++

average = sum/3

Display average
```

Flowchart:

1.2 Number Conversions

1. 1980_{10} to binary, hexadecimal and octal

To Binary:

$1980/2 = 990$	0
$990/2 = 495$	0
$495/2 = 247$	1
$247/2 = 123$	1
$123/2 = 61$	1
$61/2 = 30$	1
$30/2 = 15$	0
$15/2 = 7$	1
$7/2 = 3$	1
$3/2 = 1$	1
$1/2 = 0$	1

Binary = 11110111100

To Hexadecimal:

0111	1011	1100
7	B	C

Hexadecimal = 7BC

To Octal:

011	110	111	100
3	6	7	4

Octal = 3674

2. 1001001101_2 to decimal, hexadecimal and octal

To Decimal:

$$\begin{array}{rcl} 1 * 1 & = & 1 \\ 0 * 2 & = & 0 \\ 1 * 4 & = & 4 \\ 1 * 8 & = & 8 \\ 0 * 16 & = & 0 \\ 0 * 32 & = & 0 \\ 1 * 64 & = & 64 \\ 0 * 128 & = & 0 \\ 0 * 256 & = & 0 \\ 1 * 512 & = & 512 \\ \text{TOTAL} & = & 589 \end{array}$$

Decimal = 589

To Hexadecimal:

0010	0100	1101
2	4	D

Hexadecimal = 24D

To Octal:

001	001	001	101
1	1	1	5

Octal = 1115

3. 76_8 to binary, hexadecimal and decimal

To Binary:

$$\begin{array}{c} 111 \\ 7 \end{array}$$
$$\begin{array}{c} 110 \\ 6 \end{array}$$

Binary = 111110

To Hexadecimal:

$$\begin{array}{c} 0011 \\ 3 \end{array}$$
$$\begin{array}{c} 1110 \\ E \end{array}$$

Hexadecimal = 3E

To Decimal:

$$\begin{array}{l} 6 * 1 = 6 \\ 7 * 8 = 56 \\ \text{TOTAL} = 62 \end{array}$$

Decimal = 62

4. $43F_{16}$ to binary, decimal and octal

To Binary:

4	3	F
0100	0011	1111

Binary = 010000111111

To Decimal:

$$\begin{aligned} F * 1 &= 15 \\ 3 * 16 &= 48 \\ 4 * 256 &= 1024 \\ \text{TOTAL} &= 1087 \end{aligned}$$

Decimal = 1087

To Octal:

010	000	111	111
2	0	7	7

Octal = 02077

Chapter 2 (No exercises)

Chapter 3 Exercises

3.1 Hello World!

```
/**
 * This class prints the line "Welcome to Java Programming
 [YourName]!!!"
 * on screen.
 */
public class HelloWorld
{

    public static void main(String[] args){

        System.out.println("Welcome to Java Programming
 [YourName]!!!");

    }

}
```

3.2 The Tree

```
/**
 * A program that prints four lines on screen
 */
public class TheTree
{

    public static void main(String[] args){

        System.out.println("I think I shall never see,");
        System.out.println("a poem as lovely as a tree.");
        System.out.println("A tree whose hungry mouth is
pressed");
        System.out.println("Against the Earth's flowing
breast.");

    }

}
```

Chapter 4 Exercises

4.1 Declaring and printing variables

```
/**
 * A program that declares different variables
 * then outputs the values of the variables
 */
public class VariableSample
{
    public static void main(String[] args){

        //declares integer number with 10 as initial value
        int number = 10;

        //declares character letter with 'a' as initial value
        char letter = 'a';

        //declares boolean result with true as initial value
        boolean result = true;

        //declares String str with "hello" as initial value
        String str = "hello";

        //prints the values of the variables on screen
        System.out.println("Number = "+number);
        System.out.println("letter = "+letter);
        System.out.println("result = "+result);
        System.out.println("str = "+str);
    }
}
```

4.2 Getting the average of three numbers

```
/**
 * A program that solves for the average
 * of the three numbers: 10,20, and 45
 * then outputs the result on the screen
 */
public class AverageNumber
{
    public static void main(String[] args){

        //declares the three numbers
        int num1 = 10;
        int num2 = 20;
        int num3 = 45;

        //get the average of the three numbers
        // and saves it inside the ave variable
        int ave = (num1+num2+num3)/3;

        //prints the output on the screen
        System.out.println("number 1 = "+num1);
        System.out.println("number 2 = "+num2);
        System.out.println("number 3 = "+num3);
        System.out.println("Average is = "+ave);
    }
}
```

4.3 Output greatest value

```
/**
 * A program that outputs the number with
 * the greatest value given three numbers
 */
public class GreatestValue
{
    public static void main(String[] args){

        //declares the numbers
        int num1 = 10;
        int num2 = 23;
        int num3 = 5;
        int max = 0;

        //determines the highest number
        max = (num1>num2)?num1:num2;
        max = (max>num3)?max:num3;

        //prints the output on the screen
        System.out.println("number 1 = "+num1);
        System.out.println("number 2 = "+num2);
        System.out.println("number 3 = "+num3);
        System.out.println("The highest number is = "+max);
    }
}
```

4.4 Operator precedence

1. $((a/b)^c)^{(d-e+f-(g*h))+i)}$
2. $(((((3*10)*2)/15)-2+4)^2)^2$
3. $(r^{(((s*t)/u)-v)+w})^{(x-(y++))})$

Chapter 5 Exercises

5.1 Last 3 words (BufferedReader version)

```
import java.io.*;

/**
 * A program that asks three words from the user
 * and then prints it on the screen as a phrase
 */
public class LastThreeWords
{
    public static void main(String[] args){

        //declares the variable reader as the
        BufferedReader reader = new BufferedReader( new
                                                    InputStreamReader( System.in));

        //declares the String variables for the three words
        String firstWord = "";
        String secondWord = "";
        String thirdWord = "";

        try{

            System.out.print("Enter word1: ");
            firstWord = reader.readLine();//gets the 1st word

            System.out.print("Enter word2: ");
            secondWord = reader.readLine();//gets the 2nd word

            System.out.print("Enter word3: ");
            thirdWord = reader.readLine();//gets the 3rd word

        }catch( IOException e){
            System.out.println("Error in getting input");
        }

        //prints the phrase
        System.out.println(firstWord + " " + secondWord
                           + " " + thirdWord);
    }
}
```

5.2 Last 3 words (JOptionPane version)

```
import javax.swing.JOptionPane;

/**
 * A program that asks three words from the user using the
 * JOptionPane
 * and then displays these three words as a phrase on the
 * screen
 */
public class LastThreeWords
{
    public static void main(String[] args){

        //gets the first word from the user
        String firstWord = JOptionPane.showInputDialog
            ("Enter word1");
        //gets the second word from the user
        String secondWord = JOptionPane.showInputDialog
            ("Enter word2");
        //gets the third word from the user
        String thirdWord = JOptionPane.showInputDialog
            ("Enter word3");

        //displays the message
        JOptionPane.showMessageDialog(null, firstWord+
            " "+secondWord+
            " "+thirdWord);
    }
}
```


Chapter 6 Exercises

6.1 Grades

Using BufferedReader:

```
import java.io.*;

/**
 * Gets three number inputs from the user
 * then displays the average on the screen
 */
public class Grades
{

    public static void main(String[] args){

        //declares the variable reader as the
        //BufferedReader reader = new BufferedReader
        //    ( new InputStreamReader( System.in));

        int firstGrade = 0;
        int secondGrade = 0;
        int thirdGrade = 0;
        double average = 0;

        try{
            System.out.print("First grade: ");
            firstGrade = Integer.parseInt
                (reader.readLine());

            System.out.print("Second grade: ");
            secondGrade = Integer.parseInt
                (reader.readLine());

            System.out.print("Third grade: ");
            thirdGrade = Integer.parseInt
                (reader.readLine());

        }catch( Exception e){
            System.out.println("Input is invalid");
            System.exit(0);
        }

        //solves for the average
        average = (firstGrade+secondGrade+thirdGrade)/3;

        //prints the average of the three exams
        System.out.print("Average: "+average);

        if(average>=60)    System.out.print(" ;-");
        else               System.out.print(" ;-(");

    }
}
```

Using JOptionPane:

```
import javax.swing.JOptionPane;

/**
 * Gets three number inputs from the user
 * then displays the average on the screen
 */
public class Grades
{
    public static void main(String[] args){

        double firstGrade = 0;
        double secondGrade = 0;
        double thirdGrade = 0;
        double average = 0;

        try{
            firstGrade = Double.parseDouble
                (JOptionPane.showInputDialog
                    ("First grade"));

            secondGrade = Double.parseDouble
                (JOptionPane.showInputDialog
                    ("Second grade"));

            thirdGrade = Double.parseDouble
                (JOptionPane.showInputDialog
                    ("Third grade"));

        }catch( Exception e){
            JOptionPane.showMessageDialog(null,
                "Input is invalid");
            System.exit(0);
        }

        //solves for the average
        average = (firstGrade+secondGrade+thirdGrade)/3;

        if(average>=60){
            JOptionPane.showMessageDialog
                (null,"Average :
                    "+average+" ;-");
        }
        else{
            JOptionPane.showMessageDialog
                (null,"Average :
                    "+average+" ;-");
        }
    }
}
```

6.2 Number in words

Using if-else statement:

```
import javax.swing.JOptionPane;

/**
 * Transforms a number input from 1-10 to words
 * using if-else
 */
public class NumWords
{

    public static void main(String[] args){

        String msg = "";
        int input = 0;

        //gets the input string
        input = Integer.parseInt(JOptionPane.showInputDialog
                                ("Enter number"));

        //sets msg to the string equivalent of input
        if(input == 1)      msg = "one";
        else if(input == 2)  msg = "two";
        else if(input == 3)  msg = "three";
        else if(input == 4)  msg = "four";
        else if(input == 5)  msg = "five";
        else if(input == 6)  msg = "six";
        else if(input == 7)  msg = "seven";
        else if(input == 8)  msg = "eight";
        else if(input == 9)  msg = "nine";
        else if(input == 10) msg = "ten";
        else                 msg = "Invalid number";

        //displays the number in words if within range
        JOptionPane.showMessageDialog(null,msg);
    }
}
```

Using switch statement:

```
import javax.swing.JOptionPane;

/**
 * Transforms a number input from 1-10 to words
 * using switch.
 */
public class NumWords
{
    public static void main(String[] args){

        String msg = "";
        int input = 0;

        //gets the input string
        input = Integer.parseInt
(JOptionPane.showInputDialog
    ("Enter number"));

        //sets msg to the string equivalent of input
        switch(input){
            case 1:
                msg = "one";
                break;
            case 2:
                msg = "two";
                break;
            case 3:
                msg = "three";
                break;
            case 4:
                msg = "four";
                break;
            case 5:
                msg = "five";
                break;
            case 6:
                msg = "six";
                break;
            case 7:
                msg = "seven";
                break;
            case 8:
                msg = "eight";
                break;
            case 9:
                msg = "nine";
                break;
            case 10:
                msg = "ten";
                break;
            default:
                msg = "Invalid number";
                break;
        }
        //displays the number in words if within range
        JOptionPane.showMessageDialog(null,msg);
    }
}
```

6.3 Hundred Times

Using while-loop:

```
import java.io.*;

/**
 * A program that prints a given name one hundred times
 * using while loop
 */
public class HundredNames{

    public static void main(String[] args){

        BufferedReader reader = new BufferedReader
            (new InputStreamReader
             ( System.in));

        String name = "";
        int counter = 0;

        //gets the users' name
        try{
            System.out.print("Enter name: ");
            name = reader.readLine();
        }catch(Exception e){
            System.out.println("Invalid input");
            System.exit(0);
        }

        //while loop that prints the name one hundred times
        while(counter < 100){
            System.out.println(name);
            counter++;
        }
    }
}
```

Using do-while loop:

```
import java.io.*;

/**
 * A program that prints a given name one hundred times
 * using do-while loop
 */
public class HundredNames
{
    public static void main(String[] args){

        BufferedReader reader = new BufferedReader
                                (new InputStreamReader
                                 ( System.in));

        String name = "";
        int counter = 0;

        //gets the users' name
        try{
            System.out.print("Enter name: ");
            name = reader.readLine();
        }catch(Exception e){
            System.out.println("Invalid input");
            System.exit(0);
        }

        //do-while loop that prints the name one hundred
times
        do{
            System.out.println(name);
            counter++;
        }while(counter < 100);
    }
}
```

Using for loop:

```
import java.io.*;

/**
 * A program that prints a given name one hundred times
 * using do-while loop
 */
public class HundredNames
{
    public static void main(String[] args){

        BufferedReader reader = new BufferedReader
            (new InputStreamReader
             ( System.in));

        String name = "";

        //gets the users' name
        try{
            System.out.print("Enter name: ");
            name = reader.readLine();
        }catch(Exception e){
            System.out.println("Invalid input");
            System.exit(0);
        }

        //for loop that prints the name one hundred times
        for(int counter = 0; counter < 100; counter++){
            System.out.println(name);
        }
    }
}
```

6.4 Powers

Using while-loop:

```
import javax.swing.JOptionPane;

/**
 * Computes the power of a number given the base and the
 * exponent. The exponent is limited to positive numbers only.
 */
public class Powers
{
    public static void main(String[] args){

        int base = 0;
        int exp = 0;
        int power = 1;
        int counter = 0;

        //gets the user input for base and power using
        // JOptionPane
        base = Integer.parseInt
            (JOptionPane.showInputDialog("Base"));
        exp = Integer.parseInt
            (JOptionPane.showInputDialog("Exponent"));

        //limits the exp to positive numbers only
        if(exp < 0 ){
            JOptionPane.showMessageDialog
                (null,"Positive numbers only please");
            System.exit(0);
        }

        //while loop that solves for the power
        while(counter < exp){
            power = power*base;
            counter++;
        }

        //displays the result
        JOptionPane.showMessageDialog
            (null,base+" to the "+exp+
                " is "+power);
    }
}
```


Using do-while loop:

```
import javax.swing.JOptionPane;

/**
 * Computes the power of a number given the base and the
 * exponent. The exponent is limited to positive numbers only.
 */
public class Powers
{
    public static void main(String[] args){

        int base = 0;
        int exp = 0;
        int power = 1;
        int counter = 0;

        //gets the user input for base and power
        //using JOptionPane
        base = Integer.parseInt(JOptionPane.showInputDialog
                                ("Base"));
        exp = Integer.parseInt(JOptionPane.showInputDialog
                                ("Exponent"));

        //limits the exp to positive numbers only
        if(exp < 0 ){
            JOptionPane.showMessageDialog
                (null,"Positive numbers only please");
            System.exit(0);
        }

        //do-while loop that solves the power given the base
        // and exponent
        do{
            if(exp != 0)
                power = power*base;
            counter++;
        }while(counter < exp);

        //displays the result
        JOptionPane.showMessageDialog(null,base +
                                    " to the "+exp
                                    + " is "+power);
    }
}
```

Using for loop:

```
import javax.swing.JOptionPane;

/**
 * Computes the power of a number given the base and the
 * exponent. The exponent is limited to positive numbers only.
 */
public class Powers
{
    public static void main(String[] args){

        int base = 0;
        int exp = 0;
        int power = 1;
        int counter = 0;

        //gets the user input for base and power using
        // JOptionPane
        base = Integer.parseInt(JOptionPane.showInputDialog
                                ("Base"));
        exp = Integer.parseInt(JOptionPane.showInputDialog
                                ("Exponent"));

        //limits the exp to positive numbers only
        if(exp < 0 ){
            JOptionPane.showMessageDialog(null,"Positive
                                         numbers only please");
            System.exit(0);
        }

        //for loop for computing the power
        for(counter = 0; counter < exp; counter++){
            power = power*base;
        }

        //displays the result
        JOptionPane.showMessageDialog(null,base +
                                     " to the "+exp + " is "+power);
    }
}
```

Chapter 7 Exercises

7.1 Days of the Week

Using while loop:

```
/**
 * Uses an array string to save the days of the wee
 * then prints it on the screen.
 */
public class DaysOfTheWeek
{

    public static void main(String[] args){

        //declares the String array of the days of the week
        String[] days = {"Sunday", "Monday", "Tuesday",
                        "Wednesday", "Thursday", "Friday",
                        "Saturday"};

        int counter = 0;

        //while loop that prints the days of the week
        while(counter < days.length){
            System.out.println(days[counter]);
            counter++;
        }
    }
}
```

Using do-while loop:

```
/**
 * Uses an array string to save the days of the wee
 * then prints it on the screen with a do-while loop.
 */
public class DaysOfTheWeek
{

    public static void main(String[] args){

        //declares the String array of the days of
        // the week
        String[] days = {"Sunday", "Monday", "Tuesday",
                        "Wednesday", "Thursday", "Friday",
                        "Saturday"};

        int counter = 0;

        //do-while loop that prints the days of the
        // week
        do{
            System.out.println(days[counter]);
            counter++;
        }while(counter < days.length);
    }
}
```

Using for loop:

```
/**
 * Uses an array string to save the days of the wee
 * then prints it on the screen with a for loop.
 */
public class DaysOfTheWeek
{
    public static void main(String[] args){

        //declares the String array of the days of
        // the week
        String[] days={"Sunday","Monday","Tuesday",
            "Wednesday","Thursday","Friday",
            "Saturday"};

        //for loop that prints the days of the week
        for(int counter = 0; counter < days.length;
counter++)
            System.out.println(days[counter]);
    }
}
```

7.2 Greatest number

```
import javax.swing.JOptionPane;

/**
 * A program that uses JOptionPane to get ten numbers
 * from the user then outputs the largest number.
 */
public class GreatestNumber
{
    public static void main(String[] args){

        int[] num = new int[10];
        int counter;
        int max = 0;

        //for loop that gets the 10 numbers from the user
        for(counter = 0; counter < 10; counter++){

            num[counter] = Integer.parseInt
                (JOptionPane.showInputDialog(
                    "Enter number "+(counter+1)));

            //gets the maximum number
            if((counter == 0)|| (num[counter] > max))
                max = num[counter];
        }

        //displays the number with the greatest number
        JOptionPane.showMessageDialog(null,
            "The number with the greatest
            value is "+max);
    }
}
```

Chapter 8 Exercises

8.1 Print Arguments

```
/**
 * A program that prints the string from the command line if
 * any.
 */
public class CommandLineSample
{
    public static void main(String[] args){
        //checks if a command line argument exists
        if(args.length == 0)
            System.exit(0);

        //for loop that prints the arguments from the
        //command line
        for(int counter=0;counter<args.length; counter++){
            System.out.println(args[counter]);
        }
    }
}
```

Chapter 9 Exercises

9.1 Defining terms

See definitions in book.

9.2 Java Scavenger Hunt

To the teacher: These are just **some sample methods** in the Java API that you can use. Check the Java API for more answers.

Sample Usage:

```
public class Homework1
{
    public static void main(String []args){

        //1. endsWith
        String str = "Hello";
        System.out.println( str.endsWith( "slo" ) );

        //2. forDigit
        System.out.println( Character.forDigit(13, 16) );

        //4. floor
        System.out.println( Math.floor(3.14));

        //5. isDigit
        System.out.println( "0=" + Character.isDigit('0'));
        System.out.println( "A=" +Character.isDigit('A'));

        //3.
        System.exit(1);
        System.out.println("if this is executed, exit was
                           not called");
    }
}
```

Class and Method declaration:

1. Class: String
Method: public boolean endsWith(String suffix)
2. Class: Character
Method: public static char forDigit(int digit, int radix)
3. Class: System
Method: public static void exit(int status)
4. Class: Math
Method: public static double floor(double a)
5. Class: Character
Method: public static boolean isDigit(char ch)

Chapter 10 Exercises

10.1 Address Book Entry

```
/**
 * An address book class that record a persons
 * name, address, telephone number, and email address
 */
public class AddressBookEntry
{

    private String name;
    private String add;
    private int tel;
    private String email;

    /**
     * default constructor
     */
    public AddressBookEntry(){
        name = "";
        add = "";
        tel = 0;
        email = "";
    }

    /**
     * Creates an AddressBookEntry object with the given
     * name, address, telephone number and email adress
     */
    public AddressBookEntry(String name, String add,
                             int tel, String email){
        this.name = name;
        this.add = add;
        this.tel = tel;
        this.email = email;
    }

    /**
     * returns the variable name
     */
    public String getName(){
        return name;
    }

    /**
     * changes the variable name
     */
    public void changeName(String name){
        this.name = name;
    }

    /**
     * returns the variable add
     */
    public String getAddress(){
        return add;
    }
}
```

```
    /**
     * changes the variable add
     */
    public void changeAddress(String add){
        this.add = add;
    }

    /**
     * returns the variable tel
     */
    public int getTelNumber(){
        return tel;
    }

    /**
     * changes the variable tel
     */
    public void changeTelNumber(int tel){
        this.tel = tel;
    }

    /**
     * returns the variable email
     */
    public String getEmailAdd(){
        return email;
    }

    /**
     * changes the variable email
     */
    public void changeEmailAdd(String email){
        this.email = email;
    }
}
```


10.2 AddressBook

```
import java.io.*;

/**
 * Creates an addressbook that contains 100 AddressBookEntries
 */
public class AddressBook
{
    //index of the last entry
    private int top = 0;

    //constant number that indicates the maximum
    //number of entries in the address book
    private static final int MAXENTRIES = 100;

    //array of Address Book Entries
    private AddressBookEntry[] list;

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

        BufferedReader keyIn = new BufferedReader
            (new InputStreamReader
            (System.in));
        AddressBook addBook = new AddressBook();
        String act = "";

        while(true){

            //displays the options
            System.out.println("\n[A] Add entry");
            System.out.println("[D] Delete entry");
            System.out.println("[V] View all entries");
            System.out.println("[U] Update entry");
            System.out.println("[Q] Quit");
            System.out.print("Enter desired action: ");

            try{
                //gets the choice
                act = keyIn.readLine();

            }catch(Exception e){
                System.out.println("Error");
            }
        }
    }
}
```

```
        //checks for the appropriate action for
        // his choice
        if (act.equals("A") || act.equals("a"))
            addBook.addEntry();
        else if (act.equals("D") || act.equals("d"))
            addBook.delEntry();
        else if (act.equals("V") || act.equals("v"))
            addBook.viewEntries();
        else if (act.equals("U") || act.equals("u"))
            addBook.updateEntry();
        else if (act.equals("Q") || act.equals("q"))
            System.exit(0);
        else
            System.out.println
                ("Unknown command");
    }
}
```

```
/**
 * creates the AddressBook
 */
public AddressBook(){
    list = new AddressBookEntry[MAXENTRIES];
}

/**
 * method for adding an AddressBookEntry to the Addressbook
 */
public void addEntry(){

    BufferedReader keyIn = new BufferedReader
        (new InputStreamReader
         (System.in));

    String      name = "";
    String      add = "";
    int         tel = 0;
    String      email = "";

    if(top == MAXENTRIES){
        System.out.println("Address Book is full");
        return;
    }

    //asks the user for the data of the address book
    try{
        System.out.print("Name:           ");
        name = keyIn.readLine();
        System.out.print("Address:         ");
        add = keyIn.readLine();
        System.out.print("Telephone number: ");
        tel = Integer.parseInt(keyIn.readLine());
        System.out.print("Email Address:    ");
        email = keyIn.readLine();
    }catch(Exception e){
        System.out.println(e);
        System.exit(0);
    }

    AddressBookEntry entry = new AddressBookEntry
        (name, add, tel, email);
    list[top] = entry;
    top++;
}
```

```
/**
 * method that deletes an AddressBookEntry from the
 * Addressbook with the index
 */
public void delEntry(){

    BufferedReader keyIn = new BufferedReader
        (new InputStreamReader(System.in));
    int index = 0;

    //checks if the address book is empty
    if(top == 0){
        System.out.println("Address Book is empty");
        return;
    }

    //asks for the entry which is to be deleted
    try{
        //shows the current entries on the record book
        viewEntries();

        System.out.print("\nEnter entry number: ");
        index = Integer.parseInt(keyIn.readLine())-1;
    }catch(Exception e){}

    //checks if the index is within bounds
    if(index < 0 || index >= top){
        System.out.println("Index Out Of Bounds");
        return;
    }else{
        for( int i=index; i<top; i++ ){
            list[i] = list[i+1];
        }
        list[top] = null;
        top--;
    }
}

/**
 * method that prints all the entries in the AddressBook
 */
public void viewEntries(){

    for(int index = 0; index < top; index++){
        System.out.println((index+1)+" Name:"+
            list[index].getName());
        System.out.println("Address:"+
            list[index].getAddress());
        System.out.println("Telephone Number:"+
            list[index].getTelNumber());
        System.out.println("Email Address:"+
            list[index].getEmailAdd());
    }
}
```

```
/**
 * method that updates an entry
 */
public void updateEntry(){

    BufferedReader keyIn = new BufferedReader(new
InputStreamReader
                                (System.in));
    int index = 0;
    String name = "";
    String add = "";
    int tel = 0;
    String email = "";

    //asks for the entries data
    try{
        System.out.print("Entry number:      ");
        index =Integer.parseInt(keyIn.readLine())-1;
        System.out.print("Name:              ");
        name = keyIn.readLine();
        System.out.print("Address:           ");
        add = keyIn.readLine();
        System.out.print("Telephone number: ");
        tel = Integer.parseInt(keyIn.readLine());
        System.out.print("Email Address:     ");
        email = keyIn.readLine();
    }catch(Exception e){
        System.out.println(e);
        System.exit(0);
    }

    //updates the entry
    AddressBookEntry entry = new AddressBookEntry
                                (name, add, tel, email);
    list[index] = entry;
}
}
```

Chapter 11 Exercises

11.1 Extending StudentRecord

```
/**
 * An object that holds the data for a student
 */
public class StudentRecord
{
    protected String      name;
    protected String      address;
    protected int         age;
    protected double      mathGrade;
    protected double      englishGrade;
    protected double      scienceGrade;
    protected double      average;

    protected static int studentCount;

    /**
     * Returns the name of the student
     */
    public String getName(){
        return name;
    }

    /**
     * Changes the name of the student
     */
    public void setName(String temp){
        name = temp;
    }

    /**
     * Returns the address of the student
     */
    public String getAddress(){
        return address;
    }

    /**
     * Changes the address of the student
     */
    public void setAddress(String temp){
        address = temp;
    }

    /**
     * Returns the age of the student
     */
    public int getAge(){
        return age;
    }

    /**
     * Changes the age of the student
     */
    public void setAge(int temp){
        age = temp;
    }
}
```

```
    /**
     * Returns the englishGrade of the student
     */
    public double getEnglishGrade(){
        return englishGrade;
    }

    /**
     * Changes the englishGrade of the student
     */
    public void setEnglishGrade(double temp){
        englishGrade = temp;
    }

    /**
     * Returns the mathGrade of the student
     */
    public double getMathGrade(){
        return mathGrade;
    }

    /**
     * Changes the mathGrade of the student
     */
    public void setMathGrade(double temp){
        mathGrade = temp;
    }

    /**
     * Returns the scienceGrade of the student
     */
    public double getScienceGrade(){
        return scienceGrade;
    }

    /**
     * Changes the scienceGrade of the student
     */
    public void setScienceGrade(double temp){
        scienceGrade = temp;
    }

    /**
     * Computes the average of the english, math and
     * science grades
     */
    public double getAverage(){
        return (mathGrade+englishGrade+scienceGrade)/3;
    }

    /**
     * Returns the number of instances of the
     * StudentRecords
     */
    public static int getStudentCount(){
        return studentCount;
    }
}
```

```
/**
 * A student record for a Computer Science student
 */
public class ComputerScienceStudentRecord extends StudentRecord
{
    private String    studentNumber;
    private double    comSciGrade;

    /**
     * Returns the studentNumber of the student
     */
    public String getStudentNumber(){
        return studentNumber;
    }

    /**
     * Changes the studentNumber of the student
     */
    public void setStudentNumber(String temp){
        studentNumber = temp;
    }

    /**
     * Returns the comSciGrade of the student
     */
    public double getComSciGrade(){
        return comSciGrade;
    }

    /**
     * Changes the comSciGrade of the student
     */
    public void setComSciGrade(double temp){
        comSciGrade = temp;
    }
}
```


11.2 Abstract Classes

```
/**
 * Definition of shape abstract class
 */
public abstract class Shape
{
    /**
     * returns the area of a certain shape
     */
    public abstract double getArea();

    /**
     * returns the name of the shape
     */
    public abstract String getName();
}

/**
 * Class definition for object circle
 */
public class Circle extends Shape
{
    private static final double pi = 3.1416;
    private double radius = 0;

    /**
     * Constructor
     */
    public Circle(double r){
        setRadius( r );
    }

    /**
     * returns area
     */
    public double getArea(){
        return pi*radius*radius;
    }

    /**
     * returns shape name
     */
    public String getName(){
        return "circle";
    }

    /**
     * set radius
     */
    public void setRadius(double r){
        radius = r;
    }

    /**
     * returns radius
     */
    public double getRadius(){
        return radius;
    }
}
```

```
/**
 * Class definition for object square
 */
public class Square extends Shape
{
    private double    side = 0;

    /**
     * Constructor
     */
    public Square(double s){
        setSide( s );
    }

    /**
     * returns area
     */
    public double getArea(){
        return side*side;
    }

    /**
     * returns shape name
     */
    public String getName(){
        return "square";
    }

    /**
     * set length of side
     */
    public void setSide(double s){
        side = s;
    }

    /**
     * returns length of one side
     */
    public double getSide(){
        return side;
    }
}
```

Chapter 12 Exercises

12.1 Catching Exceptions 1

```
public class TestExceptions{
    public static void main( String[] args ){
        try{
            for( int i=0; true; i++ ){
                System.out.println("args["+i+"]="+args[i]);
            }
        }catch( ArrayIndexOutOfBoundsException e ){
            System.out.println("Exception caught:");
            System.out.println("    "+e);
            System.out.println("Quiting...");
        }
    }
}
```

12.2 Catching Exceptions 2

Here are three sample programs that we did before, wherein we included some exception handling.

```
import javax.swing.JOptionPane;

/**
 * A program that uses JOptionPane to get ten numbers
 * from the user then outputs the largest number.
 */
public class GreatestNumber
{
    public static void main(String[] args){

        int[] num = new int[10];
        int counter;
        int max = 0;

        //for loop that gets the 10 numbers from the user
        for(counter = 0; counter < 10; counter++){

            try{
                num[counter] = Integer.parseInt
                    (JOptionPane.showInputDialog
                        ("Enter number "+(counter+1)));
            }catch(NumberFormatException e){
                JOptionPane.showMessageDialog
                    (null,"Error "+e);
            }

            //gets the maximum number
            if((counter == 0) || (num[counter] > max))
                max = num[counter];
        }

        //displays the number with the greatest number
        JOptionPane.showMessageDialog
            (null,"The number with the greatest value is "+max);
    }
}
```

```
import javax.swing.JOptionPane;

/**
 * Transforms a number input from 1-10 to words using switch.
 */
public class NumWords
{
    public static void main(String[] args){
        String msg = "";
        int input = 0;

        try{
            //gets the input string
            input = Integer.parseInt
                (JOptionPane.showInputDialog
                 ("Enter number"));
        }catch(Exception e){
            JOptionPane.showMessageDialog(null,"Invalid
input");
            System.exit(0);
        }

        //sets msg to the string equivalent of input
        switch(input){
            case 1:
                msg = "one";
                break;
            case 2:
                msg = "two";
                break;
            case 3:
                msg = "three";
                break;
            case 4:
                msg = "four";
                break;
            case 5:
                msg = "five";
                break;
            case 6:
                msg = "six";
                break;
            case 7:
                msg = "seven";
                break;
            case 8:
                msg = "eight";
                break;
            case 9:
                msg = "nine";
                break;
            case 10:
                msg = "ten";
                break;
            default:
                msg = "Invalid number";
                break;
        }
        //displays the number in words if with in range
        JOptionPane.showMessageDialog(null,msg);
    }
}
```

```
    }
}
import javax.swing.JOptionPane;

/**
 * Computes the power of a number given the base and the
 * exponent.
 * The exponent is limited to positive numbers only.
 */
public class Powers
{

    public static void main(String[] args){

        int base = 0;
        int exp = 0;
        int power = 1;
        int counter = 0;

        //gets the user input for base and power using
        JOptionPane
        try{
            base = Integer.parseInt
                (JOptionPane.showInputDialog
                    ("Base"));
            exp = Integer.parseInt
                (JOptionPane.showInputDialog
                    ("Exponent"));
        }catch(NumberFormatException e){
            JOptionPane.showMessageDialog
                (null,"Input Error");
            System.exit(0);
        }

        //limits the exp to positive numbers only
        if(exp < 0 ){
            JOptionPane.showMessageDialog
                (null,"Positive numbers only please");
            System.exit(0);
        }

        //for loop for computing the power
        for(;counter < exp;counter++){
            power = power*base;
        }

        //displays the result
        JOptionPane.showMessageDialog
            (null,base+" to the "+exp
            +" is "+power);
    }
}
```

Appendix D : Machine Problems

Machine Problem 1: Phone Book

Write a program that will create an phonebook, wherein you can **add** entries in the phonebook, **delete** entries, **view all** entries and **search** for entries. In viewing all entries, the user should have a choice, whether to view the entries in **alphabetical** order or in **increasing** order of telephone numbers. In searching for entries, the user should also have an option to search entries **by name** or **by telephone** numbers. In searching by name, the user should also have an option if he/she wants to search by first name or last name.

MAIN MENU

- 1 - Add phonebook entry
- 2 - Delete phonebook entry
- 3 - View all entries
 - a - alphabetical order
 - b - increasing order of telephone numbers
- 4 - Search entries
 - a - by name
 - b - by telephone number
- 5 - Quit

The following will appear when one of the choices in the main menu is chosen.

Add phonebook entry

Enter Name:

Enter Telephone number:

(* if entry already exists, warn user about this)

View all entries

Displays all entries in alphabetical order

Displays all entries in increasing order of telephone #s

Search entries

Search phonebook entry by name

Search phonebook entry by telephone number

Quit

close phonebook

Machine Problem 2: Minesweeper

This is a one player game of a simplified version of the popular computer game minesweeper. First, the user is asked if he or she wants to play on a 5x5 grid or 10x10 grid. You have two 2-dimensional arrays that contains information about your grid. An entry in the array should either contain a 0 or 1. A 1 signifies that there is a bomb in that location, and a 0 if none.

For example, given the array:

```
int bombList5by5[][]={{0, 0, 1, 0, 0},
                      {0, 0, 0, 0, 0},
                      {0, 1, 0, 0, 0},
                      {0, 0, 0, 1, 1},
                      {0, 1, 1, 0, 0}};
```

Given the bomb list, we have 6 bombs on our list. The bombs are located in (row,col) cells, (0,2), (2,1), (3,3), (3,4), (4,1) and (4,2).

If the user chooses a cell that contains a bomb, the game ends and all the bombs are displayed. If the user chooses a cell that does not contain a bomb, a number appears at that location indicating the number of neighbors that contain bombs. The game should end when all the cells that do not contain bombs have been marked (player wins) or when the user steps on a bomb(player loses).

Here's a sample output of the game, given the bombList5by5.

```
Welcome to Minesweeper!
Choose size of grid (Press 1 for 5x5, Press 2 for 10x10): 1
[ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ]
Enter row and column of the cell you want to open[row col]: 1 1
[ ][ ][ ][ ][ ]
[ ][2][ ][ ][ ]
[ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ]
[ ][ ][ ][ ][ ]
Enter row and column of the cell you want to open[row col]: 3 2
[ ][ ][ ][ ][ ]
[ ][2][ ][ ][ ]
[ ][ ][ ][ ][ ]
[ ][ ][4][ ][ ]
[ ][ ][ ][ ][ ]
Enter row and column of the cell you want to open[row col]: 0 2
[ ][ ][ ][ ][ ]
[ ][2][ ][ ][ ]
[ ][X][ ][ ][ ]
[ ][ ][4][ ][ ]
[ ][ ][ ][ ][ ]
Ooppps! You stepped on a bomb. Sorry, game over!
```

Machine Problem 3: Number Conversion

Create your own scientific calculator that will convert the inputted numbers to the four number representations (Decimal, Binary, Octal, Hexadecimal). Your program should output the following menu on screen.

```
MAIN MENU:
Please type the number of your choice:
1 - Binary to Decimal
2 - Decimal to Octal
3 - Octal to Hexadecimal
4 - Hexadecimal to Binary
5 - Quit
```

The following will appear when one of the choices in the main menu is chosen.

```
Choice 1:
Enter a binary number: 11000
11000 base 2 = 24 base 10
(goes back to main menu)

Choice 2:
Enter a Decimal number: 24
24 base 10 = 30 base 8
(goes back to main menu)

Choice 3:
Enter an Octal number: 30
30 base 8 = 18 base 16
(goes back to main menu)

Choice 4:
Enter a Hexadecimal number: 18
18 base 16 = 11000 base 2

Choice 1:
Enter a binary number: 110A
Invalid binary number!
Enter a binary number: 1
1 base 2 = 1 base 10
(goes back to main menu)

(user chooses 5)
Goodbye!
```

You can be more creative with your user interface if you want to, as long as the program outputs the correct conversion of numbers.

References

1. Programming Language. From Wikipedia at http://en.wikipedia.org/wiki/Programming_language
2. Programming Language. From Webopedia at http://www.webopedia.com/TERM/p/programming_language.html
3. Programming Language. From Answers.com at <http://www.answers.com/topic/programming-language>
4. High-Level Programming Language. From Wikipedia at http://en.wikipedia.org/wiki/High-level_programming_language
5. Defining Flowchart Symbols. Available at http://www.patton-patton.com/basic_flow_chart_symbols.htm
6. Integrated Development Environment. From Webopedia at http://www.webopedia.com/TERM/I/integrated_development_environment.html
7. Variables and Expressions. Available at <http://www.geocities.com/SiliconValley/Park/3230/java/jav1002.html>
8. Writing Abstract Classes and Methods. Available at <http://java.sun.com/docs/books/tutorial/java/javaOO/abstract.html>
9. Defining an Interface. Available at <http://java.sun.com/docs/books/tutorial/java/interpack/interfaceDef.html>
20. Inheritance and Polymorphism. Available at <http://home.cogeco.ca/~ve3ll/jatutor7.htm>
21. The Essence of OOP using Java, Runtime Polymorphism through Inheritance. Available at <http://www.developer.com/tech/article.php/983081>
22. Gary B. Shelly, Thomas J. Cashman, Joy L. Starks. Java Programming Complete Concepts and Techniques. Course Technology Thomson Learning. 2001.
23. Stephen J. Chapman. Java for Engineers and Scientists 2nd Edition. Pearson Prentice Hall. 2004
24. Deitel & Deitel. Java How to Program 5th Edition.
25. Sun Java Programming Student Guide SL-275. Sun Microsystems. February 2001.