УНИВЕРСИТЕТ ИМЕНИ СУЛЕЙМАНА ДЕМИРЕЛЯ

Сатабалдиев А. Б.

JAVA PROGRAMMING

Методическая разработка

Каскелен, 2016

УДК 004. 4(075.8)
ББК 32.973я73
S 24

Рекомендовано к изданию Учебно-методическим советом Университета имени Сулеймана Демиреля протокол №1 от 25.08.2015 г.

Рецензенты:

Акшабаев А., PhD, Казахстанско-Британский технический университет
Жапаров М.К., PhD, Университет имени Сулеймана Демиреля

Сатабалдиев А. Б.

S 24 Java Programming: Методическая разработка. — Каскелен, Университет имени Сулеймана Демиреля, 2016. — 60 с. — на английском языке.

Satabaldiyev A. B.

Java Programming: Methodical working-out. — Kaskelen, Suleyman Demirel University, 2016. — 60 p. — In English.

ISBN 978-601-7537-24-1

Методическая разработка «Java Programming» предназначена для бакалавров по специальностям 5В03000 — «Информационные Системы» и 5В040000 — «Вычислительная Техника и Программное Обеспечение».

Methodical working-out « Java Programming » is designed for bachelors of specialties 5B030000 — “Information Systems” and 5B040000 — “Computing Systems and Software”.

УДК 004. 4(075.8)
ББК 32.973я73

ISBN 978-601-7537-24-1

© Сатабалдиев А. Б.
© Университет имени Сулеймана Демиреля, 2016

CHAPTER ONE

Java History. Introduction to Simple Java Program

Java (programming language)

Java is a programming language originally developed by James Gosling at Sun Microsystems (which has since merged into Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java applications are typically compiled to bytecode (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture. Java is a general-purpose, concurrent, class-based, object-oriented language that is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere", meaning that code that runs on one platform does not need to be edited to run on another. Java is currently one of the most popular programming languages in use, particularly for client-server web applications, with a reported 10 million users**.**

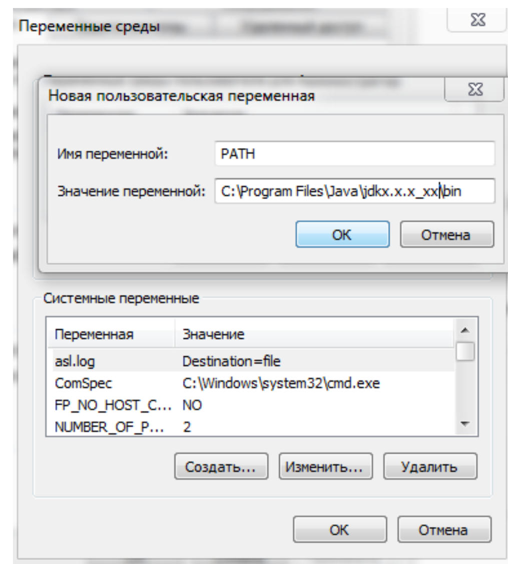
Microprocessors are having a profound impact in intelligent consumer-electronic devices. Recognizing this, Sun Microsystems in 1991 funded an internal corporate research project code-named Green, which resulted in the development of a C++ - based language that its creator, James Gosling, called Oak after an oak tree outside his window at Sun. It was later discovered that there already was a computer language called Oak. When a group of Sun people visited a local coffee shop, the name Java was suggested, and it stuck.

Installing the Java Development Kit on Windows*:*

Java is a programming language that allows programs to be written that can then be run on more than one type of operating system. A program written in Java can run on Windows, UNIX, Linux etc. as long as there is a Java runtime environment installed. In order to develop our own program we need some library JDK and JRE.

1. Download JDK and JRE. (http://www.oracle.com/technetwork/java/javase/downloads/index.html)
2. Install. Be sure that installation finished successfully.
3. Change class path. Open Computer > Properties> Advanced > Environment Variables > Create
4. Enter to Variable name "PATH" and Variable value "java_directory" . Like "C:\Program Files\Java\jdk7.1.0_22\bin"
5. Click OK.
6. Open command line (cmd)
7. Type "javac". Is it working?

   It will look like:



Installing the Java Development Kit on Ubuntu*:*

1. *You should have internet connection.*
2. Open command line (Ctrl+Alt+T)
3. And type there:

```
sudo add-apt-repository ppa:ferramroberto/java
sudo apt-get update
sudo apt-get install sun-java6-jdk sun-java6-plugin
```

Note: if you get any error then try after follow command

```
sudo add-apt-repository ppa:ferramroberto/java
```

then run this

```
apt-get install python-software-properties
sudo add-apt-repository ppa:ferramroberto/java
```

To write your first program, you'll need:

1. **The Java SE Development Kit (JDK).** You can download and install to your PC (Windows, Ubuntu or Mac) from this link http://docs.oracle.com/javase/7/docs/webnotes/install/
2. **A text editor**
   In this example, we'll use Notepad, a simple editor included with the Windows platforms. You can easily adapt these instructions if you use a different text editor. These two items are all you'll need to write your first application.

Creating Your First Application

Your first application, `HelloWorldAppication`, will simply display the greeting "Hello world!". To create this program, you will:

- **Create a source file** A source file contains code, written in the Java programming language, that you and other programmers can understand. You can use any text editor to create and edit source files.
- **Compile the source file into a ".class" file**
  The Java programming language *compiler* (javac) takes your source file and translates its text into instructions that the Java virtual machine can understand. The instructions contained within this file are known as *bytecodes*.
- **Run the program**
  The Java application uses the Java virtual machine to run your application.

Create a Source File

To create a source file, you have two options:

- You can save the file HelloWorldApp.java on your computer and avoid a lot of typing. Then, you can go straight to Compile the Source File into a .class File.
- Or, you can use the following (longer) instructions.

First, start your editor. You can launch the Notepad editor from the **Start** menu by selecting **Programs > Accessories > Notepad**. In a new document, type in the following code:

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

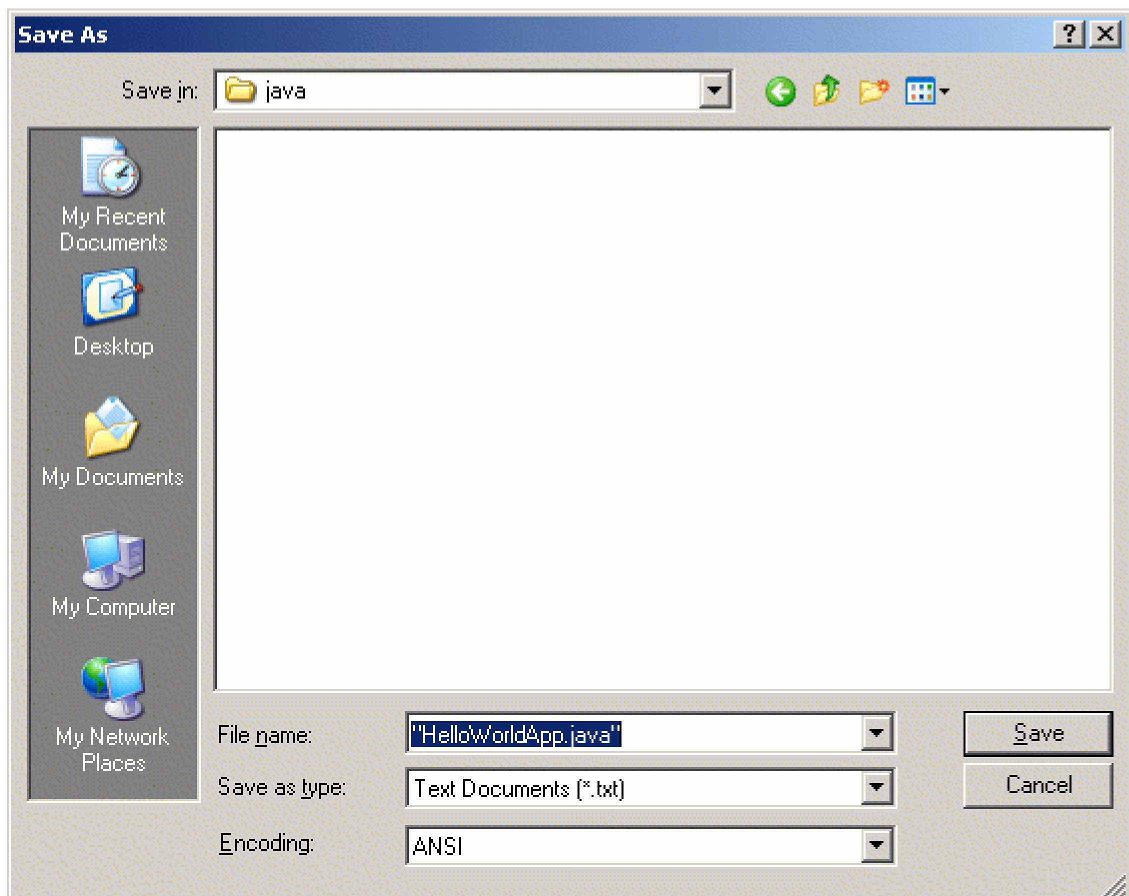
Be Careful When You Type N

`HelloWorldApp` is *not* the same as `helloworldapp`.

Save the code in a file with the name `HelloWorldApp.java`. To do this in Notepad, first choose the **File** > **Save As** menu item. Then, in the **Save As** dialog box:

1. Using the **Save in** combo box, specify the folder (directory) where you'll save your file. In this example, the directory is `java` on the `C` drive.
2. In the **File name** text field, type `"HelloWorldApp.java"`, including the quotation marks.
3. From the **Save as type** combo box, choose **Text Documents (*.txt)**.
4. In the **Encoding** combo box, leave the encoding as ANSI.

When you're finished, the dialog box should look like this.

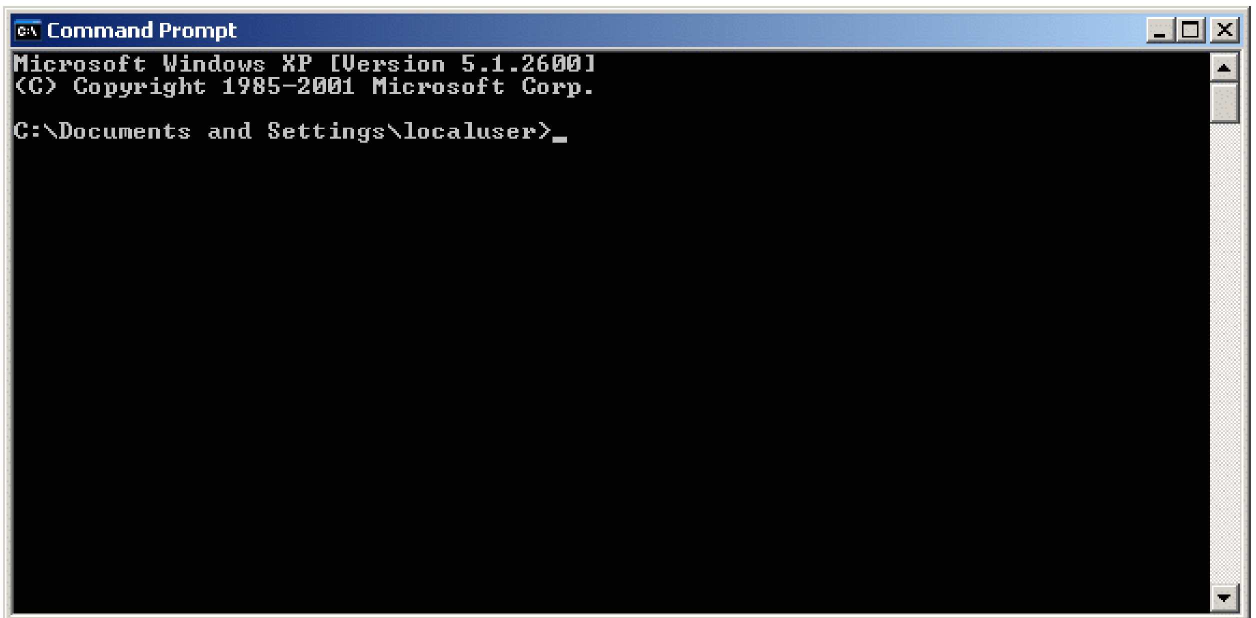


The Save As dialog just before you click **Save**.

Now click **Save**, and exit Notepad.

Compile the Source File into a .class File

Bring up a shell, or "command," window. You can do this from the Start menu by choosing Command Prompt (Windows XP), or by choosing Run... and then entering cmd. The shell window should look similar to the following figure.



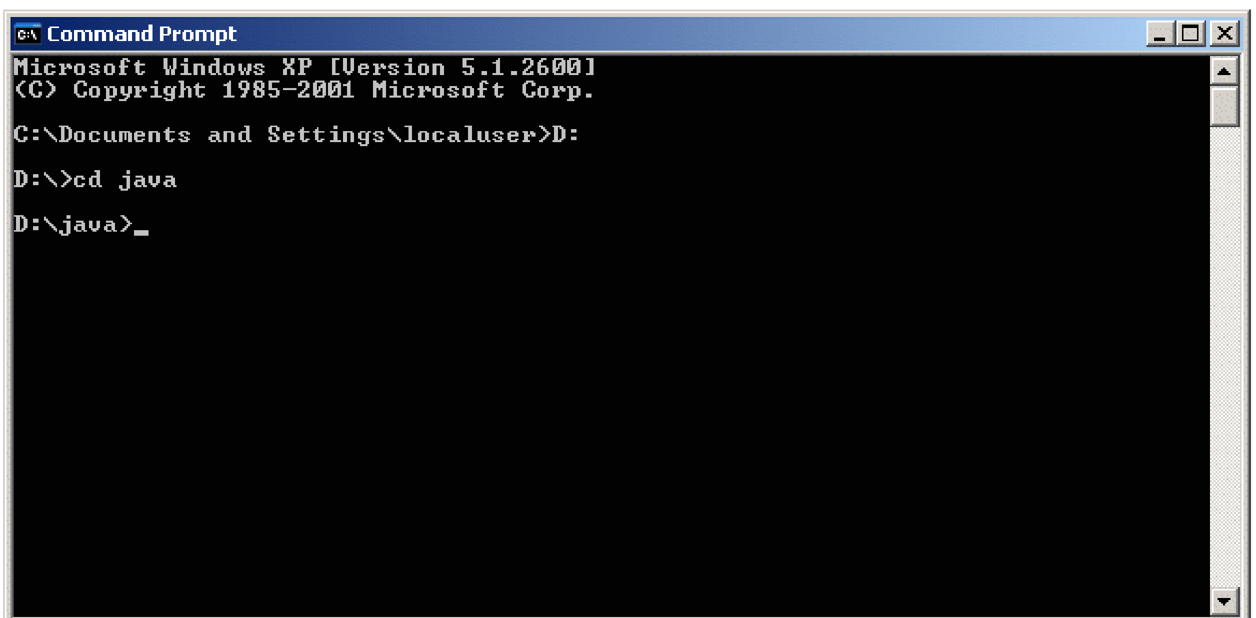
A shell window.

The prompt shows your *current directory*. When you bring up the prompt, your current directory is usually your home directory for Windows XP (as shown in the preceding figure.

To compile your source file, change your current directory to the directory where your file is located. For example, if your source directory is java on the C drive, type the following command at the prompt and press **Enter**:

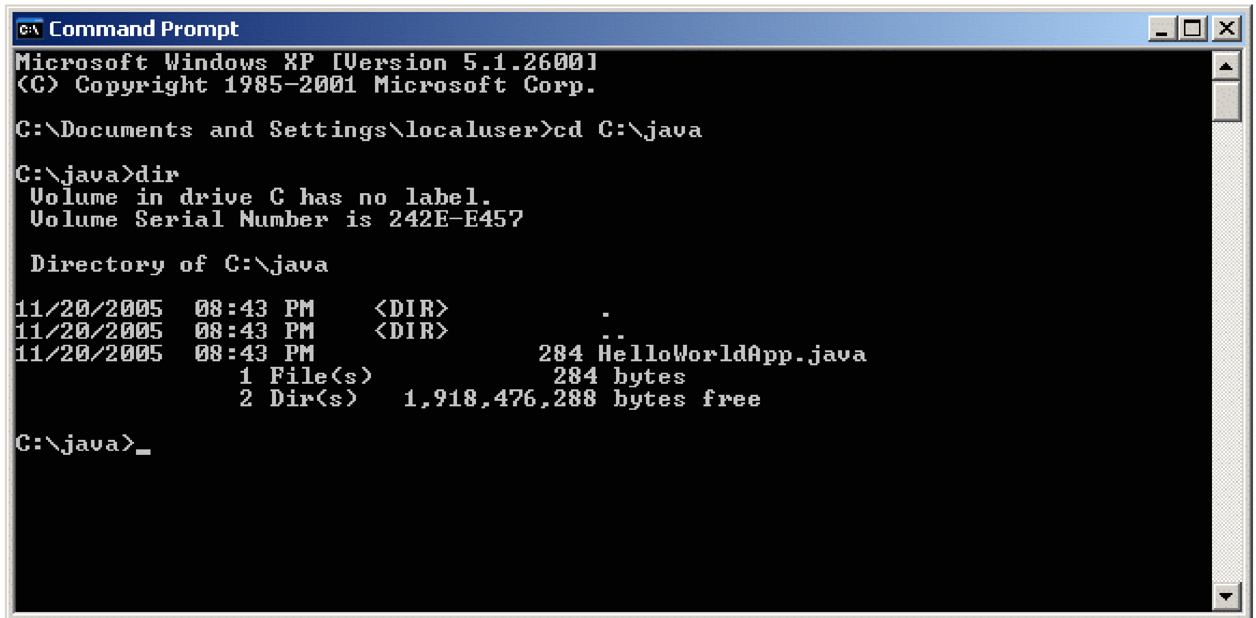
cd C:\java

Now the prompt should change to C:\java>. **Note:** To change to a directory on a different drive, you must type an extra command: the name of the drive. For example, to change to the java directory on the D drive, you must enter D:, as shown in the following figure.



Changing directory on an alternate drive.

If you enter dir at the prompt, you should see your source file, as the following figure shows.

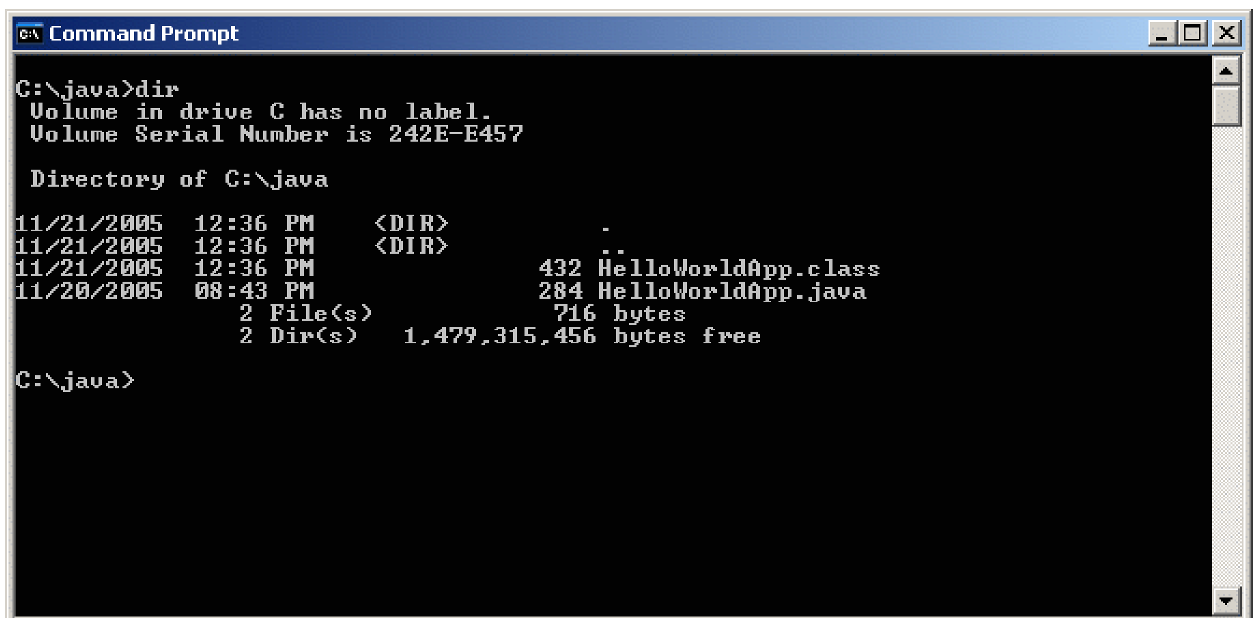


Directory listing showing the .java source file.

Now you are ready to compile. At the prompt, type the following command and press **Enter**.

javac HelloWorldApp.java

The compiler has generated a bytecode file, HelloWorldApp.class. At the prompt, type dir to see the new file that was generated, as shown in the following figure.



Directory listing, showing the generated .class file

Now that you have a .class file, you can run your program.

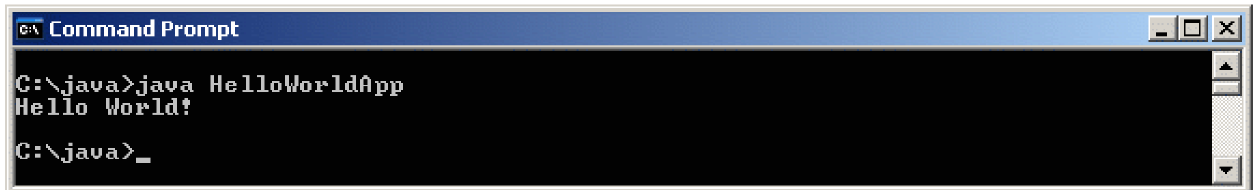
If you encounter problems with the instructions in this step, consult the Common Problems (and Their Solutions).

Run the Program

In the same directory, enter the following command at the prompt:

java HelloWorldApp

The next figure shows what you should now see:



The program prints "Hello World!" to the screen.

Congratulations! Your program works!

CHAPTER TWO

Variables and the Primitive Types

Variables:

Programs manipulate data that are stored in memory. In machine language, data can only be referred to by giving the numerical address of the location in memory where it is stored. In a high-level language such as Java, names are used instead of numbers to refer to data. It is the job of the computer to keep track of where in memory the data is actually stored; the programmer only has to remember the name. A name used in this way — to refer to data stored in memory — is called a variable.

In Java, the only way to get data into a variable—that is, into the box that the variable names—is with an **assignment statement** . An **assignment statement** takes the form:

{ variable } = { expression } ;

where { expression } represents anything that refers to or computes a data value.

The Primitive Types and String:

There are eight so-called **primitive types** built into Java. The primitive types are named : *byte*, *short*, *int*, *long*, *float*, *double*, *char*, and *boolean*. The first four types hold integers (whole numbers such as 7, -38477, and 0). The four integer types are distinguished by the ranges of integers they can hold. The float and double types hold real numbers (such as 3.6 and -145.99). Again, the two real types are distinguished by their range and accuracy. A variable of type char holds a single character from the Unicode character set. And a variable of type boolean holds one of the two logical values **true** or **false**.

1. A variable of type byte holds a string of eight bits, which can represent any of the integers between -128 and 127, inclusive.
   byte b = 1204;
2. A variable of type short corresponds to two bytes (16 bits). Type short have values in the range -32768 to 32767.
   short s = 100;
3. A variable of type int corresponds to four bytes (32 bits). Type int have values in the range -2147483648 to 2147483647.
   int i = 20;
4. A variable of type long corresponds to eight bytes (64 bits). Type long have values in the range -9223372036854775808 to 9223372036854775807.
   long l = 1L;
5. A variable of type float data type is represented in four bytes of memory, using a standard method for encoding real numbers. The maximum value for a float is about 10 raised to the power 38.
   float f = 2f;
6. A double takes up 8 bytes, can range up to about 10 to the power 308.
   double d = 3.14;
7. A variable of type char occupies two bytes in memory. The value of a char variable is a single character such as *A, *, x,* or a *space character* . Variable declares by single-quote.
   char c = ‘A’, char c= ‘\u0101’;
8. For the type boolean, there are precisely two literals: true and false.
   boolean b = false;

There is a fundamental difference between the **primitive types** and the **String** type: Values of type String are objects. While we will not study objects in detail in next chapters, it will be useful for you to know a little about them and about a closely related topic: *classes*. String declares by double-quotes.

String s = ‘Hello students! I’m a String’;

The Scanner Class

The **Scanner class** was introduced in Java 5.0 to make it easier to read basic data types from a character input source. It does not solve the problem completely, but it is a big improvement. The Scanner class is in the package **java.util** .

Scanner standardInputScanner = new Scanner(System.in);
System.in – allows input from command line

Examples:

Section 1.

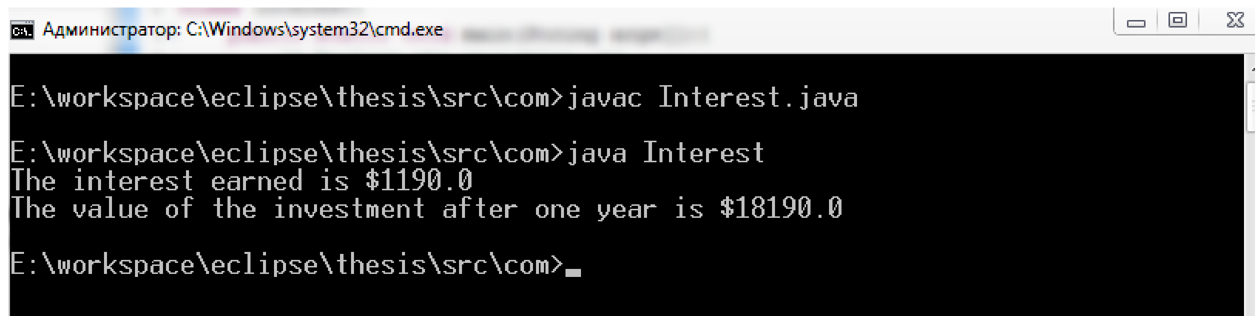
Exercise 1:

Code:
```
package com;
class Interest{
	public static void main(String args[]){
		/* Declare the variables. */
		double principal; // The value of the investment.
		double rate; // The annual interest rate.
		double interest; // Interest earned in one year.

		/* Initialize variables. */
		principal = 17000;
		rate = 0.07;
		interest = principal * rate; // Compute the interest.
		principal = principal + interest;
		// Compute value of investment after one year, with interest.
		// (Note: The new value replaces the old value of principal.)
		/* Output the results. */
		System.out.print("The interest earned is $");
		System.out.println(interest);
		System.out.print("The value of the investment after one year is
$");
		System.out.println(principal);
	} // end of main()
} // end of class Interest
```

Solve together:

This class implements a simple program that will compute the amount of interest that is earned on $17,000 invested at an interest rate of 0.07 for one year. The interest and the value of the investment after one year are printed to standard output.

Output:



Exercise 2:

Code:

```
package com;
import java.util.Scanner;
public class ScannerTest {
	public static void main(String args [])
	{
		/*Declare variables*/
		String name;
		// create class Scanner
		Scanner input = new Scanner(System.in);
		System.out.println("Enter your name : ");
		// wait until user enters his/her name
		name = input.nextLine();
		System.out.println("How are you "+name+"?");
	} //end of main()
} // end of ScannerTest
```

Solve together:

In this program, class will implement Scanner class from library **java.util** as we told before. Program will ask user to input his / her name from command line (System.in). Then will wait until enter button presses. When user enter Enter button, displays output shown below.

Output:

```
Администратор: C:\Windows\system32\cmd.exe

E:\workspace\eclipse\thesis\src\com>javac ScannerTest.java

E:\workspace\eclipse\thesis\src\com>java ScannerTest
Enter your name :
Aydin
How are you Aydin?

E:\workspace\eclipse\thesis\src\com>_
```

<u>Section 2.</u>

Task.

1. Write a program that will print your initials to standard output in letters that are nine lines tall. Each big letter should be made up of a bunch of *'s. For example, if your initials were "DJE", then the output would look something like:

```
******           *************        **********
**    **                **            **
**     **               **            **
**       **             **            **
**       **             **            ********
**       **      **     **            **
**     **         **    **            **
**    **           **   **            **
*****               ****              **********
```

2. Write a program that asks the user's name, and then greets the user by name. Before outputting the user's name, convert it to upper case letters. For example, if the user's name is Aydin, then the program should respond "Hello, AYDIN, nice to meet you!".
3. Write an application that asks user to enter two integers, obtains them from the user and prints their sum, product, difference and quotient (division).
4. Given two strings, a and b, print the result of putting them together in the order abba, e.g. "Hi" and "Bye" returns "HiByeByeHi".
5. Given three ints, a b c, display true if it is possible to add two of the ints to get the third.
6. Given three ints, a b c, return true if one of b or c is "close" (differing from a by at most 1), while the other is "far", differing from both other values by 2 or more. Note: Math.abs(num) computes the absolute value of a number.

input	output
1,2,10	True
1,2,3	False
4,1,3	True

7. Implement a Java-main-method that prints out the multiplication table for all numbers from 1 to 10. Use the tabulator character '\t' to align the values. The output of your method should be as follows:

```
1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100
```

8. Implement a Java-method with three local int variables a, b, and c that sorts these three values in ascending order by comparing and exchanging their values. At the end of the program a <= b <= c must hold.
9. Implement a Java-method that prints out the day of the week for a given day (1..31), month (1..12) and year.

CHAPTER THREE

Algorithm, Searching Algorithms

Algorithm:

A program is an expression of an idea. A programmer starts with a general idea of a task for the computer to perform. Presumably, the programmer has some idea of how to perform the task by hand, at least in general outline. The problem is to flesh out that outline into a complete, unambiguous, step-by-step **procedure** for carrying out the task. Such a **procedure** is called an "algorithm." (Technically, an algorithm is an unambiguous, step-by-step procedure that terminates after a finite number of steps; we don't want to count procedures that go on forever.) An algorithm is not the same as a program.

Finally, a program is written in some particular programming language. An algorithm is more like the idea behind the program, but it's the idea of the steps the program will take to perform its task, not just the idea of the task itself. An algorithm can be expressed in any language, including English.

Pseudocode **:**

Is an informal language that helps programmers develop algorithms without having to worry about the strict details of Java language syntax. The pseudocode we present is particularly useful for developing algorithms that will be converted to structured portions of Java programs. Pseudocode is similar to everyday English. It is convenient and user friendly, but it is not an actual computer programming language.

Pseudocode does not execute on computers. Rather, it helps the programmer "think out" a program before attempting to write it in a programming language, such as Java. This chapter provides several examples of how to use pseudocode to develop Java programs.

if Single-Selection Statement

Programs use selection statements to choose among alternative courses of action. For example, suppose that the passing grade on an exam is 60. The pseudocode statement

```
If student's grade is greater than or equal to 60 then
    Print "Passed"
```

In java code it looks like:

```
if ( studentGrade >= 60 )
        System.out.println( "Passed" );
```

if...else *Double-Selection Statement*

The `if` single-selection statement performs an indicated action only when the condition is **`true`**; otherwise, the action is skipped. The `if...else` double-selection statement allows the programmer to specify an action to perform when the condition is true and a different action when the condition is false. For example, the pseudocode statement

```
If student's grade is greater than or equal to 60
    Print "Passed"
```

Else
    Print "Failed"

In java code it looks like

```
if ( grade >= 60 )
       System.out.println( "Passed" );
    else
       System.out.println( "Failed" );
```

Increment (++) and Decrement (--) Operators

Java provides two unary operators for adding 1 to or subtracting 1 from the value of a numeric variable. These are the unary increment operator, `++`, and the unary decrement operator, `--`. A program can increment by 1 the value of a variable called `v` using the increment operator, `++`, rather than the expression `v = v + 1` or `v += 1`. You can see examples from table shown below:

Operator	Expression	Explanation
++	++a	Increment `a` by `1`, then use the new value of `a` in the expression in which `a` resides.
--	b--	Use the current value of `b` in the expression in which `b` resides, then decrement `b` by `1`.

Precedence rules:

If you use several operators in one expression, and if you don't use parentheses to explicitly indicate the order of evaluation, then you have to worry about the precedence rules that determine the order of evaluation.

Precedence table

Unary operators	++, -- , ! , unary - and +
Multiplication and division	*, /, %
Addition and subtraction	+, -
Relational operators	<, >, <=, >=
Equality and inequality	==, !=
Boolean and	&&
Boolean or	||
Conditional operator	? :
Assignment operators	=, +=, -=, *=, /=, %=

Searching Algorithms. Binary Search:

A binary search algorithm is a technique for finding a particular value in a linear array, by ruling out half of the data at each step. A binary search finds the median, makes a comparison to determine whether the desired value comes before or after it, and then searches the remaining half in the same manner. A binary search is an example of a divide and conquer algorithm (more specifically a decrease and conquer algorithm) and a dichotomic search. There are many Searching Algorithms like Linear Search, Binary Search and etc. We will show only Binary Search with pseudocode and java code.

- <u>Pseudocode:</u>

```
Assume sorted data M(0) … M(n)
Binsearch (x)
        left = 0
        right = n
        midpoint = (left + right) / 2
        WHILE x notequal M(midpoint)
                midpoint = (left + right) / 2
                IF M(midpoint) < x THEN left = midpoint
                                        ELSE right = midpoint
        END WHILE
```

- <u>Java Code:</u>

```
public int binarySearch(int[] search, int find) {
                int start, end, midPt;
                start = 0;
                end = search.length - 1;
                while (start <= end) {
                        midPt = (start + end) / 2;
                        if (search[midPt] == find) {
                                return midPt;
                        } else if (search[midPt] < find) {
                                start = midPt + 1;
                        } else {
                                end = midPt - 1;
                        }
                }
                return -1;
        }
```

Examples:

<u>Section 1.</u>

Exercise 1:

Code:

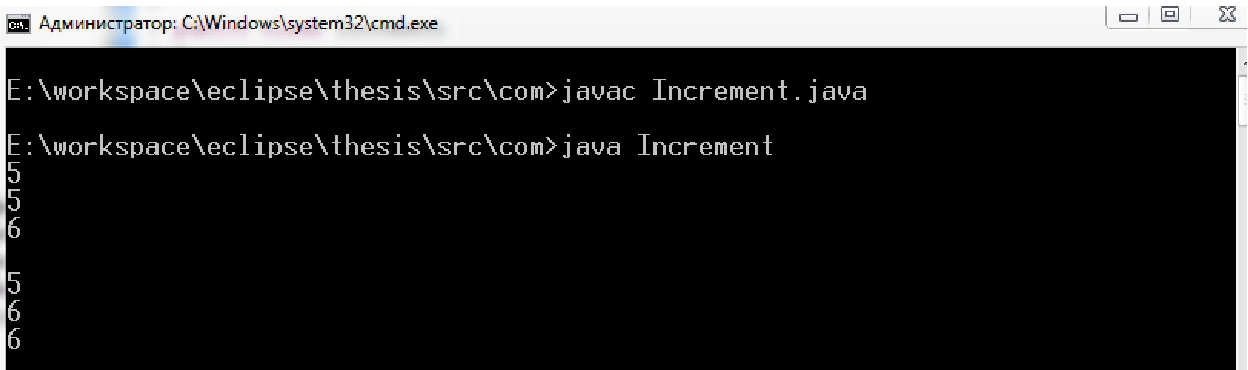
```
public class Increment
{
    public static void main( String args[] )
   {
        int c;
        // demonstrate postfix increment operator
        c = 5; // assign 5 to c
        System.out.println( c );   // print 5
        System.out.println( c++ ); // print 5 then postincrement
        System.out.println( c );   // print 6

        System.out.println(); // skip a line

        // demonstrate prefix increment operator
        c = 5; // assign 5 to c
```

```
        System.out.println( c );    // print 5
        System.out.println( ++c ); // preincrement then print 6
        System.out.println( c );    // print 6

    } // end main
}
```

Solve together:

In this example, we simply want to show the mechanics of the ++ operator, so we use only one class declaration containing method main. Occasionally, when it does not make sense to try to create a reusable class to demonstrate a simple concept, we will use a mechanical example contained entirely within the **main** method of a single class.

Output:



Section 2

Tasks

1. Write an application that uses only the output statements

```
System.out.print( "* " );
System.out.print( " " );
System.out.println();
```

to display the checkerboard pattern that follows. Note that a `System.out.println` method call with no arguments causes the program to output a single newline character.

```
E:\workspace\eclipse\ExtraLesson\src\task>java drawAsterics
* * * * * * * * * *
 * * * * * * * * * *
* * * * * * * * * *
 * * * * * * * * * *
* * * * * * * * * *
 * * * * * * * * * *
* * * * * * * * * *
 * * * * * * * * * *
* * * * * * * * * *
 * * * * * * * * * *
```

2.Write an application that reads three nonzero integers and determines and prints whether they could represent the sides of a right triangle.

3.A large company pays its salespeople on a commission basis. The salespeople receive $200 per week plus 9% of their gross sales for that week. For example, a salesperson who sells $5,000 worth of merchandise in a week receives $200 plus 9% of $5,000, or a total of $650. You have been supplied with a list of the items sold by each salesperson. Develop a Java application that inputs one salesperson's items sold for last week and calculates and displays that salesperson's earnings. There is no limit to the number of items that can be sold by a salesperson.

Item	Value
1	239.99
2	129.75
3	99.95
4	350.89

4. Write a Java application that uses looping to print the following table of values: (Hint: you can use printf).

N	10*N	100*N	1000*N
1	10	100	1000
2	20	200	2000
3	30	300	3000
4	40	400	4000
5	50	500	5000

5. How many times do you have to roll a pair of dice before they come up snake eyes? You could do the experiment by rolling the dice by hand. Write a computer program that simulates the experiment. The program should report the number of rolls that it makes before the dice come up snake eyes. (Note: “Snake eyes” means that both dice show a value of 1.)

CHAPTER FOUR

Java Loops. Logical Operator

In this chapter, we demonstrate Java's *for, do...while* and *switch* statements. Through a series of short examples using while and for, we explore the essentials of counter-controlled repetition.

We introduce the *break* and *continue* program control statements. We discuss Java's logical operators, which enable programmers to use more complex conditional expressions in control statements.

Finally, we summarize Java's control statements and the proven problem-solving techniques presented in previous chapters.

while Repetition Statement

A **repetition statement** allows the programmer to specify that a program should repeat an action while some condition remains **true**. The pseudocode statement:

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

for Repetition Statement

Java provides the *for* repetition statement, which specifies the counter-controlled-repetition details in a **single line** of code.

The general format of the `for` statement is

```
for  ( < initialization ;  condition;  increment >)
          < statement >
```

initialization - loop's control variable and provides its initial value;

condition - determines whether the loop should continue executing;

increment - modifies the control variable's value (increment or decrement);

statement – any java statements, which ends with semicolon;

Be careful we have also two semicolons between them.

do . . while Repetition Statement

The *do...while* repetition statement is similar to the *while* statement. The difference is, in the *while*, the program tests the loop-continuation condition at the beginning of the loop, before executing the loop's body. If the condition is false, the body **never executes**.

The *do...while* statement tests the loop-continuation condition after executing the loop's body; therefore, the body always executes **at least once**. When a *do...while* statement terminates, execution continues with the next statement in sequence.

The general format of the *do..while* loop is:

```
do
  {
    < statement >
  } while ( < condition > );
```

Also we have semicolon at the end of while statement.

switch Multiple-Selection Statement

We discussed the *if* single-selection statement and the *if...else* double-selection statement in previous chapters. Java provides the *switch* multiple-selection statement to perform different actions based on the possible values of an integer variable or expression.

The *switch* statement allows you to test the value of an expression and, depending on that value, to jump directly to some location within the switch statement. General format is:

```
switch ( <expression >)
```

```
{
        case <c-1>:
                <statements-1>;
                break;
        case <c-2>:
                <statements-2>;
                break;
        .
        .
        .
        case <c-n>:
                <statements-n>;
                break;
        default:
                <statements>;
}
```

First step, switch takes value of the expression, the c-1, c-2 (constants) depends on this value. If one of constants value satisfies with given value, switch will jump to that case, and listen to that case's command. When appears *break* statement, it will finish. { about *break* little bit later in this chapter}. And again goes to first line – switch statement.

The *default,* which provides a default jump point that, is used when the value of the expression is not satisfying in any case constants.

If we do not write *break* statement, program look for order of cases. First c-1, c-2 and so on.

Note. The *default* do not have *break* statement.

Suppose that, in our switch we have cases 1 till 10. Switch takes the value of expression 5. Our program will jump 5th case, and then works there till appearance of *break* statement. If the expression value equal to 11, program automatically jumps to *default* case.

break and *continue* Statements

Java provides statements `break` and `continue` to alter the flow of control. The preceding section showed how `break` can be used to terminate a `switch` statement's execution. This section discusses how to use `break` in a repetition statement.

In addition to the `break` and `continue` statements discussed in this section, Java provides the labeled `break` and `continue` statements for use in cases in which a programmer needs to conveniently alter the flow of control in nested control statements.

The **break** statement, when executed in a *while*, *for*, *do...while* or *switch*, causes immediate **exit** from that statement. Common uses of the **break** statement are to escape early from a loop or to **skip** the remainder of a switch.

The **continue** statement, when executed in a *while, for or do...while*, skips the remaining statements in the loop body and proceeds with the **next iteration** of the loop.

You will see examples in Examples section.

Logical Operators

Java provides **logical operators** to enable programmers to form more complex conditions by combining simple conditions. The logical operators are && (conditional AND), || (conditional OR), & (boolean logical AND), | (boolean logical inclusive OR), ^ (boolean logical exclusive OR) and ! (logical NOT). Let's talk about each of them one by one. Truth table may be useful for you (fig - 4.1).

Conditional AND (&&) Operator

Suppose that we wish to ensure at some point in a program that two conditions are **both true** before we choose a certain path of execution. In this case, we can use the **AND** operator, as follows:

```
if ( gender == FEMALE && age >= 65 )
```

This `if` statement contains two simple conditions. The condition `gender == FEMALE` compares variable `gender` to the constant `FEMALE`. This might be evaluated, for example, to determine whether a person is female. The condition `age >= 65` might be evaluated to determine whether a person is a senior citizen.

Conditional OR (||) Operator

Now suppose that we wish to ensure that **either** or **both** of two conditions are true before we choose a certain path of execution. In this case, we use the **OR** operator, as in the following program segment:

```
if ( ( semesterAverage >= 90 ) || ( finalExam >= 90 ) )
   System.out.println ( "Student grade is A" );
```

Boolean Logical AND (&) and Boolean Logical OR (|) Operators

The boolean logical AND (&) and boolean logical inclusive OR (|) operators work identically to the && (conditional AND) and || (conditional OR) operators, **with one exception**: The boolean logical operators always evaluate **both** of their operands.

For example:

Boolean Logic AND Operant:

```
if ( gender == 1 ) & ( age >= 65 )
```

In this example, program will check both, two situations. The value of *gender* and *age*. If value of *gender* is not equal to 1, program will check the value of *age*.

Conditional AND Operant:

```
if ( gender == 1 ) && ( age >= 65 )
```

In this example, program will check both, two situations. The *value* of *gender* and *age*. **But**, if the value of *gender* is not equal to 1, program will **not check** the value of *age*.

Logical Negation (!) Operator

The **! (logical NOT)** operator enables a programmer to **reverse** the meaning of a condition. Unlike the logical operators &&, ||, &, | and ^, which are binary operators that combine two conditions, the logical negation operator is a unary operator that has only a single condition as an operand.

```
boolean passed = true;
    if ( ! passed )
    System.out.printf( "You do not passed!");
```

Truth table (fig – 4.1)

X	Y	X & Y	X | Y	X ^ Y	!X
1	0	0	1	1	0
1	1	1	1	0	0
0	0	0	0	0	1
0	1	0	1	1	1

Factorial

Factorial n (usually written *n!*) is the product of all integers up to and including n (1 * 2 * 3 * ... * n). This problem is often used as a programming example, especially to show how to write recursive functions.

Examples

Section 1

Exercise 1:

Java code:

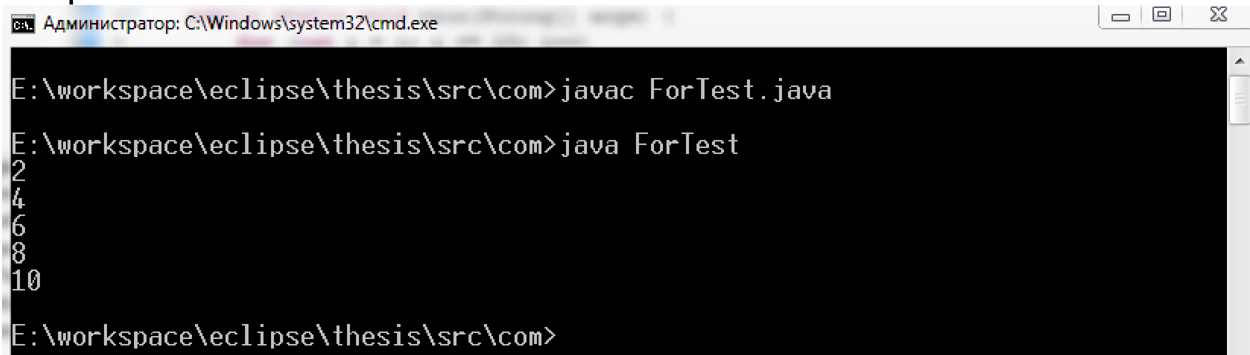
```
public class ForTest {
	public static void main(String[] args) {
		for (int i = 1; i <= 10; i++)
		{
			if(i%2==0)
			{
				System.out.println(i);
			}
		}
	}
}
```

Solve together:

Our Program prints out all even numbers from 1 till 10. This class creates *for* statement in its main method. And initialize the value of *i* to 1, then increments each time by 1 (i = i+1), while the value

of *i* less than or equal to 10. The *if* statement will check, Does the value of *i* divisible by 2 or not. If divisible, then print out this value.

Output:



Exercise 2:

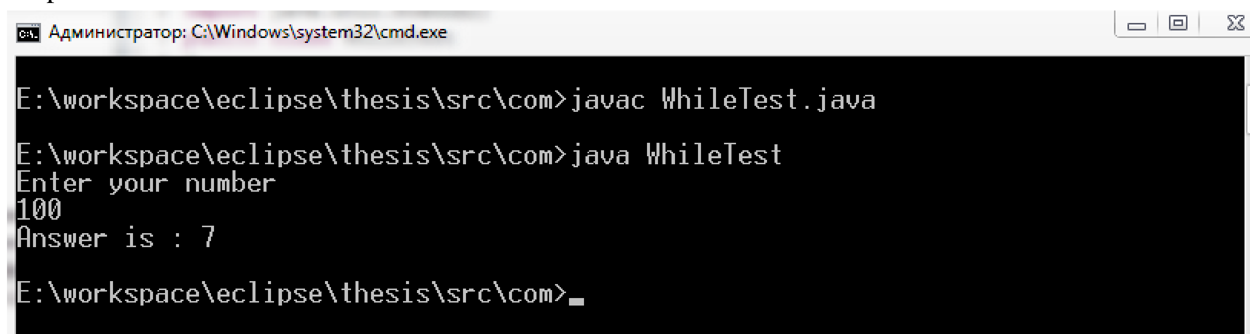
Code:

```
import java.util.Scanner;
public class WhileTest
{
	public static void main(String[] args)
	{
		Scanner in = new Scanner(System.in);
		int count = 0;    // count how many divisions we've done
		int number;
		System.out.println("Enter your number");
		number = in.nextInt();
		  while (number >= 1) {
		    number = number / 2;
		    count++;
		  }
		  System.out.println("Answer is : "+count);
	}
}
```

Solve together:

1. Here's a while loop example that uses a loop to see how man times you can divide a number by 2. Given a number, shows how many times can we divide it by 2 to get down to 1.

Output:



Exercise 3:

Java Code:

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

```
        {
                int i = 0;
                do
                {
                        if(i%2==0 && i != 0)
                        {
                                System.out.println(i);
                        }
                        i++;
                }while(i<=10);
        }
}
```

Solve together:

This program works as same as previous example (ForTest), but little bit differ. As you can see we used do .. while repetition statement. Program initialize the value of $i = 0$. Before looking for the given condition, our program checks the *if* statement. And also we have logical NOT operand and Conditional AND operand. In human words, if the value of *i* is divisible by 2 (even) AND NOT equal to 0, then print the value of *i*. Because, initial value of *i* is equal to 0. Then increment the value of *i*. The output same as ForTest.

Output:

```
Администратор: C:\Windows\system32\cmd.exe

E:\workspace\eclipse\thesis\src\com>javac DoWhileTest.java

E:\workspace\eclipse\thesis\src\com>java DoWhileTest
2
4
6
8
10

E:\workspace\eclipse\thesis\src\com>_
```

Section 2

Tasks

1. Write a simple program, that user enters his or her final grade (in digits) and your program will show result in letters. For example, if user enters 90 your program will print out “A-”. Also, if user enters less than 0 and bigger than 100 print out an exception.

A	[95-100]
A-	[90-95)
B+	[85-90)
B	[80-85)
B-	[75-80)
C+	[70-75)
C	[65-70)
C-	[60-65)
D+	[55-60)
D	[50-55)

F	[0-50)

2. Write a simple java program to calculate factorial of any given number.
3. One interesting application of computers is to display graphs and bar charts. Write an application that reads five numbers between 1 and 30. For each number that is read, your program should display the same number of adjacent asterisks. For example, if your program reads the number 7, it should display *******.
4. Write an application that prompts the user to enter the size of the side of a square, and then displays a hollow square of that size made of asterisks. Your program should work for squares of all side lengths between 1 and 20.
5. Write a program called **ComputePI** to compute the value of π, using the following series expansion. $\pi = 4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \frac{1}{15} + \cdots\right)$

6. Add all the natural numbers below one thousand that are multiples of 3 or 5. If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23. Find the sum of all the multiples of 3 or 5 below 1000.

7. What is the smallest number divisible by each of the numbers 1 to 20? 2520 is the smallest number that can be divided by each of the numbers from 1 to 10 without any remainder. What is the smallest positive number that is evenly divisible by all of the numbers from 1 to 20?

CHAPTER FIVE

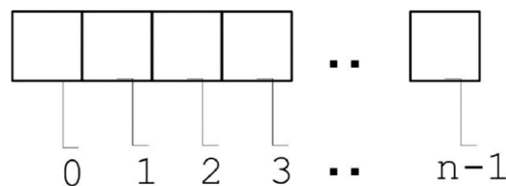
Array. Class String. Fibonacci

Arrays

An array is a data structure consisting of a **numbered** list of items, where all the items are of the same type. In Java, the items in an array are always numbered from zero up to some maximum value, which is set when the array is created.

For example, an array might contain 100 integers, numbered from 0 to 99. The items in an array can belong to one of Java's primitive types. They can also be references to objects.

Finally, array – is a set of data types. Like, set of integers, set of Strings, set of chars and etc.



Create, Initialize and Use

Array objects occupy space in memory. Like other objects, arrays are created with

keyword **new**. For example, we will create an array, set of integers which its length equal to 4. Be careful, **do not** forget two " [ ] " signs.

```
int x [] = new int [4];
```

Now we already created an array of integers. Let's fill out, initialize that array with integers.

First thing you should aware is we cannot initialize the array of integers with Strings, chars or bytes.

For example:

```
x [0] = 12;  x [1] = 15;  x [2] = -7;   x [3] = 0;
```

12	15	-7	0
0	1	2	3

If you understood how to create and declare section, you can easily use this array. Just easy. For example, let's make mathematic operations with array.

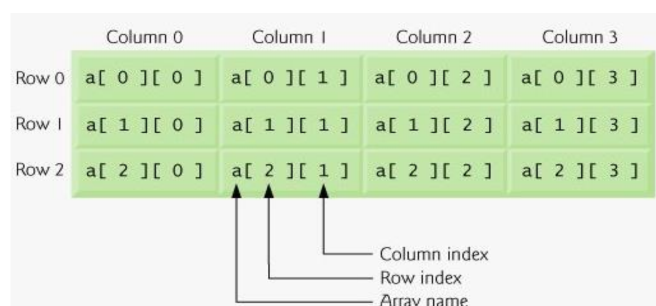
```
x [0] – x[ 2 ]  = 19; // 12 - ( - 7) = 19
x [1] * x[ 0 ]  = 180; // 12 * 15 = 180
```

Every main method has an array of Strings (String args []) in its parameters. We can automatically path arguments to this array by typing in command line "java ClassName arg0 arg1 arg2". Those arg0, arg1 and arg2 will automatically save into array args.

That is it!

Multidimensional Arrays

Multidimensional arrays with two dimensions are often used to represent **tables of values** consisting of information arranged in **rows** and **columns**. To identify a particular table element, we must specify two indices. By convention, the first identifies the element's row and the second its column. We can say that the matrix also like a two - dimensional array. It is look like:



Create, Initialize and Use

A multidimensional array with the same number of columns in every row can be created with an array-creation expression. For example, the following lines declare array `a` and assign it a reference to a two-by-two array:

```
int a[][] = new int [ 2 ][ 2 ];
```

Now we can initialize this array.

a [0][0] = 2;   a[0][1]=3;   a[1][0]=4;   a[1][1]=-7; It looks like:

2	3
4	-7

Now we can use the values of array.

a[0][1]+a[1][0]+a[1][1] = 0 // 3+4+(-7)=0

That is all about arrays!!!!

String

A value of type String is an object. That object contains data, namely the sequence of characters that make up the string. Class String is used to represent strings (texts) in Java. String – is collection of characters. Foe example:
String name = "Aydana";
String gender = "Female";
String university = new String("SDU"); // a String object

String Methods:

The String class defines a lot of functions. Here are some that you might find useful. Assume that s1 and s2 refer to values of type String:

- **s1.equals(s2)** is a function that returns a boolean value. It returns true if s1 consists of exactly the same sequence of characters as s2, and returns false otherwise.
- **s1.equalsIgnoreCase(s2)** is another boolean-valued function that checks whether s1 is the same string as s2, but this function considers upper and lower case letters to be equivalent. Thus, if s1 is "cat", then s1.equals("Cat") is false, while s1.equalsIgnoreCase("Cat") is true.
- **s1.length()**, as mentioned above, is an integer-valued function that gives the number of characters in s1.
- **s1.charAt(N)**, where N is an integer, returns a value of type char. It returns the N-th character in the string. Positions are numbered starting with 0, so s1.charAt(0) is actually the first character, s1.charAt(1) is the second, and so on. The final position is s1.length() - 1. For example, the value of "cat".charAt(1) is 'a'. An error occurs if the value of the parameter is less than zero or greater than s1.length() - 1.
- **s1.substring(N,M)**, where N and M are integers, returns a value of type String. Note that the character in position M is not included [N,M). The returned value is called a substring of s1.
- **s1.indexOf(s2)** returns an integer. If s2 occurs as a substring of s1, then the returned value is the starting position of that substring. Otherwise, the returned value is -1. You can also use s1.indexOf(ch) to search for a particular character, ch, in s1. To find the first occurrence of x at or after position N, you can use s1.indexOf(x,N).
- **s1.compareTo(s2)** is an integer-valued function that compares the two strings. If the strings are equal, the value returned is zero. If s1 is less than s2, the value returned is a number less than zero, and if s1 is greater than s2, the value returned is some number greater than zero. (If both of the strings consist entirely of lower case letters, then "less than" and "greater than" refer to alphabetical order. Otherwise, the ordering is more

complicated.)

- **s1.toUpperCase()** is a String-valued function that returns a new string that is equal to s1, except that any lower case letters in s1 have been converted to upper case. For example, "Cat".toUpperCase() is the string "CAT". There is also a function s1.toLowerCase().
- **s1.trim()** is a String-valued function that returns a new string that is equal to s1 except that any non-printing characters such as spaces and tabs have been trimmed from the beginning and from the end of the string. Thus, if s1 has the value "fred ", then s1.trim() is the string "fred".

Fibonacci: The **Fibonacci series**

0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ….

begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers. 0+1=1, 1+1=2… The Fibonacci series may be defined recursively as follows:

Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2);

Palindrome

A palindrome is a string that reads the same backwards and forwards, such as “radar”, “kazak”. The reverse() function could be used to check whether a string, word, is a palindrome by testing

“if (word.equals(reverse(word)))”.

Examples

Section 1

Exercise 1:

Java code:

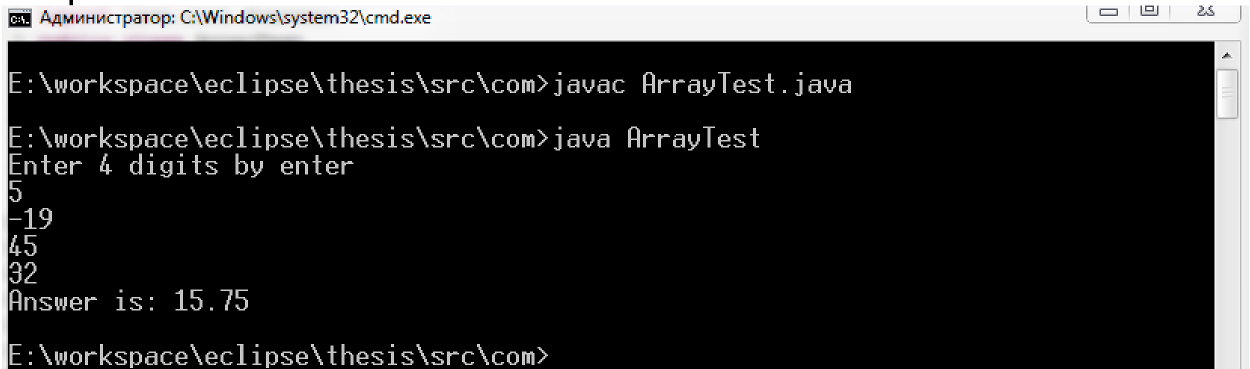
```
import java.util.Scanner;
public class ArrayTest
{
	public static void main(String[] args)
	{
		Scanner in = new Scanner(System.in);
		int m [] = new int[4];
		double sum = 0;
		double result = 0;
		System.out.println("Enter 4 digits by enter");
		for(int i=0;i<m.length;i++)
		{
			m[i] = in.nextInt();
		}
		for(int i=0;i<m.length;i++)
		{
			sum = sum+m[i];
		}
		result = sum / m.length;
		System.out.println("Answer is: "+result);
	}
```

```
}
```

Solve together:

In this example our program will ask user to input 4 numbers, then it will print their average. Remember: Average = sum of numbers / by quality.

Output:



Exercise 1:

Java code:

```
import java.util.Scanner;
public class PalindromeTest {
	public static void main(String[] args) {
		Scanner in = new Scanner(System.in);
		System.out.println("Enter your string");
		String str = in.nextLine();
		int count = 0;
		for(int i=0;i<str.length()/2;i++)
		{
			if(str.charAt(i) == str.charAt(str.length()-1-i))
			{
				count++;
			}
		}
		if(count == str.length()/2)
		{
			System.out.println("yes");
		}
		else
		{
			System.out.println("no");
		}
	}
}
```

Solve together:

Now, program will ask to input any text, that user wants to check. Does it is palindrome or not. Our loop runs till half of entered text. And compare first character with last, second with pre last and so on. If all conditions satisfies prints “yes”, “no” otherwise.

Output:

```
E:\workspace\eclipse\thesis\src\com>javac PalindromeTest.java

E:\workspace\eclipse\thesis\src\com>java PalindromeTest
Enter your string
kazak
yes

E:\workspace\eclipse\thesis\src\com>
```

Section 2

Tasks

1. Write a program that will read a sequence of positive real numbers entered by the user and will print their sum.
2. Write a program that will read a sequence of positive real numbers entered by the user and will find maximum number.
3. Suppose that we have 2 matrixes A and B. Both of them are the same size. Write a simple program that will add A and B matrixes.
4. Write a program that asks the user's name, and then greets the user by name. Before outputting the user's name, convert it to upper case letters. For example, if the user's name is Fred, then the program should respond "Hello, FRED, nice to meet you!".
5. Write a program, that will print out entered value by user in reverse order. For example "sdudent" result must "tneduds".
6. Write a program called **Fibonacci** to display the first 20 Fibonacci numbers F(n), where F(n)=F(n–1)+F(n–2) and F(1)=F(2)=1. Also compute their average. The output shall look like:

```
The first 20 Fibonacci numbers are:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
The average is 885.5
```

7. Write a program called **`PrintAnimalPattern`**, which uses `println()` to produce this pattern:

```
          '  '
           __
          (©©)
  /========\/
 / || %% ||
*  ||----||
   ¥¥    ¥¥
   ""    ""
```

Hints:

- Use escape sequence `\uhhhh` where `hhhh` are four hex digits to display Unicode characters such as ¥ and ©. ¥ is `165` (`00A5H`) and © is `169` (`00A9H`) in both ISO-8859-1 (Latin-1) and Unicode character sets.
- Double-quote (`"`) and black-slash (`\`) require escape sign inside a String. Single quote (`'`) does not require escape sign.

Print the same pattern using `printf()`. Hints: Need to use `%%` to print a `%` in `printf()` because `%` is the suffix for format specifier.

8. Write a Java program called **`Arithmetic`** that takes three command-line arguments: two integers followed by an arithmetic operator (`+`, `-`, `*` or `/`). The program shall perform the corresponding operation on the two integers and print the result. For example:

```
> java Arithmetic 3 2 +
3+2=5

> java Arithmetic 3 2 -
3-2=1

> java Arithmetic 3 2 /
3/2=1
```

CHAPTER SIX

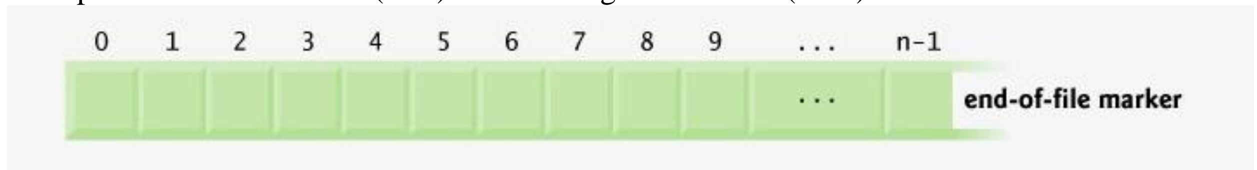
File Processing. Sort Algorithms.

Storage of data in variables and arrays is temporary. The data is lost when a local variable goes out of scope or when the program terminates. Computers use files for long-term retention of large amounts of data, even after the programs that created the data terminate.

You use files every day for tasks such as writing an essay or creating a spreadsheet. We refer to data maintained in files as **persistent data** because it exists beyond the duration of program execution. Computers store files on **secondary storage devices** such as hard disks, optical disks and magnetic tapes. In this chapter, we explain how Java programs create, update and process data files.

Files and Streams

Java views each file as a sequential **stream of bytes**. Every operating system provides a mechanism to determine the end of a file (EOF), such as an **end-of-file marker** or a count of the total bytes in the file that is recorded in a system-maintained administrative data structure. For example: At the end of text (data) there is a sign end-of-file (EOF). No more texts.



Now, we are going to show how to use the files in java. And will introduce some file operations. Java has read, write, modify and etc. operation with files. More about this operations later, in examples part. You can see some methods here.

Method	**Description**
`boolean canRead()`	Returns `true` if a file is readable by the current application; `false` otherwise.
`boolean canWrite()`	Returns `true` if a file is writable by the current application; `false` otherwise.
`boolean exists()`	Returns `true` if the name specified as the argument to the `File` constructor is a file or directory in the specified path; `false` otherwise.
`boolean isFile()`	Returns `true` if the name specified as the argument to the `File` constructor is a file; `false` otherwise.
`boolean isDirectory()`	Returns `true` if the name specified as the argument to the `File` constructor is a directory; `false` otherwise.
`boolean isAbsolute()`	Returns `true` if the arguments specified to the `File` constructor indicate an absolute path to a file or directory; `false` otherwise.
`String getAbsolutePath()`	Returns a string with the absolute path of the file or directory.
`String getName()`	Returns a string with the name of the file or directory.
`String getPath()`	Returns a string with the path of the file or directory.
`String getParent()`	Returns a string with the parent directory of the file or directory (i.e., the directory in which the file or directory can be found).
`long length()`	Returns the length of the file, in bytes. If the `File` object represents a directory, `0` is returned.
`long lastModified()`	Returns a platform-dependent representation of the time at which the file or directory was last modified. The value returned is useful only for comparison with other values returned by this method.
`String[] list()`	Returns an array of strings representing the contents of a directory. Returns `null` if the `File` object does not represent a directory.

Sort Algorithms

Selection Sort

This sorting method uses the idea of finding the biggest item in the list and moving it to the end—which is where it belongs if the list is to be in increasing order.

Once the biggest item is in its correct location, you can then apply the same idea to the remaining items. That is, find the next-biggest item, and move it into the next-to-last space, and so forth.

```
static void selectionSort(int[] A) {
	for (int lastPlace = A.length-1; lastPlace > 0; lastPlace--)
	{
		int maxLoc = 0;
		for (int j = 1; j <= lastPlace; j++)
		{
			if (A[j] > A[maxLoc])
			{
				maxLoc = j;
			}
		}
		int temp = A[maxLoc];
		A[maxLoc] = A[lastPlace];
		A[lastPlace] = temp;
	}

}
```

Insertion Sort

This method is also applicable to the problem of keeping a list in sorted order as you add new items to the list. Let's consider that case first:

Suppose you have a sorted list and you want to add an item to that list. If you want to make sure that the modified list is still sorted, then the item must be inserted into the rightlocation, with all the smaller items coming before it and all the bigger items after it. The java code looks like as follows:

```
	void insert(int[] A, int itemsInArray, int newItem) {
		int loc = itemsInArray - 1;
		while (loc >= 0 && A[loc] > newItem)
		{
			A[loc + 1] = A[loc]; // Bump item from A[loc] up to loc+1.
			loc = loc - 1; // Go on to next location.
		}
		A[loc + 1] = newItem; // Put newItem in last vacated space.
	}
```

Examples

Section 1

Exercise 1:

Java code:

```
import java.io.*;
public class FileTest
{
	public static void main(String[] args)throws IOException
	{
		File sourceFile = new File("C:/source.txt");
```

```
		File destFile = new File("E:/destination.txt");

		FileInputStream reader = new FileInputStream(sourceFile);
		PrintWriter writer = new PrintWriter(destFile);

		int eof = 0;
		String str = "";
		while((eof = reader.read()) != -1)
		{
			str+=(char)eof;
		}
		System.out.println("I read '"+str+"'");
		System.out.println("Now I'm writing to "+
destFile.getAbsolutePath());
		writer.write(str);
		writer.close();
	}
}
```

Solve together:

The source file contains “Hello guys!!!!” text.

In the main method, we declare two files. One is source file and another is destination file. Our program copied from source to destination. In the while statement, class FileInputStream reads data from source file and convert it to character then adds to String . Each character has its associated value in integer. Like: A = 65, B = 66, C = 67 and so on. “-1” end of file, no more data’s in file. Do not forget close the writers in your program.

Output:

```
Администратор: C:\Windows\system32\cmd.exe

E:\workspace\eclipse\thesis\src\com>javac FileTest.java

E:\workspace\eclipse\thesis\src\com>java FileTest
I read 'Hello guys!!!!'
Now I'm writing to E:\destination.txt

E:\workspace\eclipse\thesis\src\com>
```

Exercise 2:

Java code:

```
public class SelectionSort
{
	public static void main(String[] args)
	{
		int A [] = {4,1,7,-1,40,10,-192};
		selectionSort(A);
	}
	static void selectionSort(int[] A) {
		for (int lastPlace = A.length-1; lastPlace > 0; lastPlace--)
		{
		int maxLoc = 0;
			for (int j = 1; j <= lastPlace; j++)
			{
				if (A[j] > A[maxLoc])
				{
				maxLoc = j;
				}
```

```
				}
				int temp = A[maxLoc];
				A[maxLoc] = A[lastPlace];
				A[lastPlace] = temp;
			}
			for(int i = 0;i<A.length;i++)
			{
				System.out.print(A[i]+" ");
			}
		}
}
```

Solve together:

As we told before, array with given values, will through to selectionSort method (more about methods later), program finds biggest digit and puts the biggest value at the end of array. Then will find second biggest number and puts it in front of previous biggest value.

Output:

```
Администратор: C:\Windows\system32\cmd.exe

E:\workspace\eclipse\thesis\src\com>javac SelectionSort.java

E:\workspace\eclipse\thesis\src\com>java SelectionSort
-192 -1 1 4 7 10 40
E:\workspace\eclipse\thesis\src\com>
```

Section 2

Tasks

1. Write a program, that reads from “input.txt” two numbers (numbers entered by space) . You should make mathematical operations with them. For example: suppose we have 100 and 25 numbers in “input.txt” file. Your program must do addition, subtraction, division and multiplication operations. Also write output to “output.txt”. [Hint: String methods and StringTokenizer may useful].

2.Suppose that a file named “studentfile.txt” contains the following information: The first line of the file is the name of a student. Each of the next three lines contains an integer. The integers are the student’s scores on three exams. Write a program that will read the information in the file and display (on standard output) a message the contains the name of the student and the student’s average grade on the three exams. The average is obtained by adding up the individual exam grades and then dividing by the number of exams.

3. Suppose that a file named “searchme.txt” contains text. You should search from this file. Write a program that user will enter text from (cmd) your program searches this word from the file. If contains print “yes” otherwise “no”. [Hint: String methods and StringTokenizer may useful]
4. A file contains text. But this text encrypted. Each letter shifted by 3 to right. Write a program that will decrypt this text. Note: Space is important.
   Input: F xj zibsbo
   Output: I am clever
5. Suppose SDU students at the end of semester will have final exams. One of the lessons very difficult to them. For example Java by Mr. Satibaldiyev. Mr. Satibaldiyev stores all information in his flash drive include final exam questions. Students wants to cheat final exam questions from his flash drive during lab lesson, when he plug his flash drive in to one

of the student’s PC. In order to help SDUdents, write a program that will copy all data’s from flash drive to local disk.

6. Write a Java program called PrimeList that prompts the user for an upper bound (a positive integer), and lists all the primes less than or equal to it.

```
java WordGuess testing
Key in one character or your guess word: t
Trail 1: t__t___
Key in one character or your guess word: g
Trail 2: t__t__g
Key in one character or your guess word: e
Trail 3: te_t__g
Key in one character or your guess word: testing
Trail 4: Congratulation!
You got in 4 trials
```

7. Write a Java program called **NumberGuess** to play the number guessing game. The program shall generate a random number between 0 and 99. The player inputs his/her guess, and the program shall response with "Try higher", "Try lower" or "You got it in **n** trials" accordingly. For example:

```
> java NumberGuess
Key in your guess:
50
Try higher
70
Try lower
65
Try lower
You got it in 4 trials!
```

CHAPTER SEVEN

static Class Members. Divide and Conquer

Every object has its own copy of all the instance variables of the class. In certain cases, only one copy of a particular variable should be shared by all objects of a class. A **static field** called a **class variable** is used in such cases. A `static` variable represents classwide information all objects of the class share the same piece of data. The declaration of a `static` variable begins with the keyword `static`. For example: static String UNIVERSITY = “SDU”;

Static/Class methods

There are two types of methods.

- **Instance methods** are associated with an object and use the instance variables of that object. This is the default.
  From outside the defining class, an instance method is called by prefixing it with an *object*, which is then passed as an implicit parameter to the instance method, eg, input.nextLine();
- **Static methods** use no instance variables of any object of the class they are defined in. If you define a method to be static, you will be given a rude message by the compiler if you try to access any instance variables. You can access *static* variables, but except for constants, this is unusual. A static method is called by prefixing it with a *class name*, eg, Math.max(i,j);. Curiously, it can also be qualified with an object, which will be ignored, but the class of the object will be used.

  A *static* method can't access instance variables, it can access static variables.

Divide and Conquer - is an important algorithm design paradigm based on multi-branched recursion. A **divide** and **conquer** algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. Binary search, a divide and conquer algorithm in which the original problem is successively broken down into *single* subproblems of roughly half the original size.

Big Numbers

If the precision of the basic integer and floating-point types is not sufficient, you can turn to a couple of handy classes in the java.math package: **BigInteger** and **BigDecimal**. These are classes for manipulating numbers with an arbitrarily long sequence of digits. The **BigInteger** class implements arbitrary precision integer arithmetic, and BigDecimal does the same for floating-point numbers.

Examples:

<u>Section 1.</u>

Exercise 1:

Code:

```
import java.math.BigInteger;
public class ForFun {
	public static void main(String args[])
	{
		BigInteger bigInt1 = new BigInteger("2147483648");
		BigInteger bigInt2 = new BigInteger("2147483648");
```

```
            System.out.println("Answer is : " + bigInt1.subtract(bigInt2));
        }
}
```

Solve together:

As we told before, a variable with type int can take values only in given range [-2147483648, 2147483647]. If we enter 2147483648, then its error. “Out of range exception”. The BigInteger class can solve this type of error. As followed example, the BigInteger class takes value 2147483647 as String then make subtraction.

Output:

```
Администратор: C:\Windows\system32\cmd.exe

E:\workspace\eclipse\thesis\src\com>javac ForFun.java

E:\workspace\eclipse\thesis\src\com>java ForFun
Answer is : 0

E:\workspace\eclipse\thesis\src\com>
```

Section 2

Tasks

1. Write a simple program, that uses static variables and methods. Declare static and non-static variables and use them in static method one by one. What is a difference?
2. Write simple program, that makes mathematical operations with big integers.
3. Given an array of positive ints, return a new array of length "count" containing the first even numbers from the original array. The original array will contain at least "count" even numbers.

   {3, 2, 4, 5, 8}, 2 → {2, 4}
   {3, 2, 4, 5, 8}, 3 → {2, 4, 8}
4. Given an array of strings, return the count of the number of strings with the given length.

   {"a", "bb", "b", "ccc"}, 1 → 2
   {"a", "bb", "b", "ccc"}, 3 → 1
5. Given an array of positive ints, return a new array of length "count" containing the first endy numbers from the original array. The original array will contain at least "count" endy numbers. We'll say that a positive int n is "endy" if it is in the range 0..10 or 90..100 (inclusive).

   {9, 11, 90, 22, 6}, 2 → {9, 90}
   {9, 11, 90, 22, 6}, 3 → {9, 90, 6}
6. Given a non-empty string and an int n, print a new string where the char at index n has been removed. The value of n will be a valid index of a char in the original string (i.e. n will be in the range 0..str.length()-1 inclusive).
7. Given a non-empty string and an int N, return the string made starting with char 0, and then every Nth char of the string. So if N is 3, use char 0, 3, 6, ... and so on. N is 1 or more.

   "Miracle", 2 → "Mrce"
   "abcdefg", 2 → "aceg"

CHAPTER EIGHT

Objects, Classes and Constructors

Object-oriented programming (OOP) represents an attempt to make programs more closely model the way people think about and deal with the world. In the older styles of programming, a programmer who is faced with some problem must identify a computing task that needs to be performed in order to solve the problem.

Objects and Classes

Objects are closely related to classes. The more usual terminology is to say that objects ***belong*** to classes, but this might not be much clearer. From the point of view of programming, it is more exact to say that classes are used to create objects. A class is a kind of factory for constructing objects. The non-static parts of the class specify, or describe, what variables and subroutines the objects will contain. Objects are created and destroyed as the program runs, and there can be many objects with the same structure, if they are created using the same class.

An object that belongs to a class is said to be an instance of that class. The variables that the object contains are called **instance variables**. The subroutines that the object contains are called **instance methods.**

For example we have a class Student. The name, test1, test2, test3 are instance variables and the getAverage is instance method. And constructor is Student.

```
public class Student {
      public String name;
      public double test1, test2, test3;

      public double getAverage()
      {
            return (test1 + test2 + test3) / 3;
      }
}
```

Usage

We have created class Student already. If we want to access instance variable and methods we should create an object of this class. For example:

Student std;

However, declaring a variable does not create an object! This is an important point, which is related to this Very Important Fact:

In Java, no variable can ever hold an object.
A variable can only hold a reference to an object.

In fact, there is a special portion of memory called the heap where objects live. Instead of holding an object itself, a variable holds the information necessary to find the object in memory. This information is called a **reference** or **pointer** to the object. In effect, a reference to an object is the address of the memory location where the object is stored. When you use a variable of class type, the computer uses the reference in the variable to find the actual object. In a program, objects are created using an operator called **new**, which creates an object and returns a reference to that object. For example, assuming that std is a variable of type Student, declared as above, the assignment statement

```
std = new Student();
```

would create a new object which is an instance of the class Student, and it would store a reference to that object in the variable std. The value of the variable is a reference to the object, not the object itself.

Then we can easily use variables of Student class by std.name or std.test2. Also assign to modify another objects value by std2.name = std.name or std.test3 = 96;

Constructors

Every class has at least one constructor. If the programmer doesn't write a constructor definition in a class, then the system will provide a **default** constructor for that class. This default constructor does nothing beyond the basics: allocate memory and initialize instance variables.

If you want more than that to happen when an object is created, you can include one or more constructors in the class definition.

The definition of a constructor looks much like the definition of any other subroutine, with three exceptions.

- A constructor does not have any return type.
- The name of the constructor must be the same as the name of the class in which it is defined.
- The only modifiers that can be used on a constructor definition are the access modifiers public, private, and protected.

```
public class Student {
        public String name;
        public double test1, test2, test3;
        public Student(String name1)
        {
           name = name1;
        }
        public Student(String name1, double t1, double t2, double t3)
        {
           name = name1;
           test1 = t1;
           test2 = t2;
           test3 = t3;
        }
        public double getAverage()
        {
              return (test1 + test2 + test3) / 3;
        }
}
```

Same as previous example with class constructor. Here Student (String name1) and Student (String name1, double t1, double t2, double t3) are constructors. Difference is in their parameters. For example when we create object of class Student

```
Student s1;
s1 = new Student("Alibek");
```

Program will automatically call a constructor which has String type in its parameter. So, first constructor. If we create:

```
Student s2;
s2 = new Student("Aidana", 98.0, 99.1, 23.8);
```

then second constructor. If we create:

```
Student s3;
s3 = new Student("Askar",98.0, "Adebiet");
```

then it will an error, because we **do not have** constructor, which has String, double, String type in its parameter list.

Recursion

The programs we have discussed thus far are generally structured as methods that call one another in a disciplined, hierarchical manner. For some problems, however, it is useful to have a method call itself. Such a method is known as a **recursive method**. A recursive method can be called either directly, or indirectly through another method. Let's solve factorial with recursion method. The following code will look like :

```
public int factorial(int n) {
    if (n == 1) return 1;
    return n * factorial(n-1);
}
```

Base case: if n is 1, we can return the answer directly.

Recursion: otherwise make a recursive call with n-1 (towards the base case), i.e. call factorial(n-

Examples:

Section 1.

Exercise 1:

Java Code:

//Rectangle class

```
public class Rectangle
{
	private int width;
	private int height;
	public Rectangle(int w, int h)
	{
		width = w;
		height = h;
	}
	public int calculateArea()
	{
		int result = width*height;
		return result;
	}
	public int calculatePerimeter()
	{
		int result = 2*(width+height);
		return result;
	}
}
```

//Rectangle test class

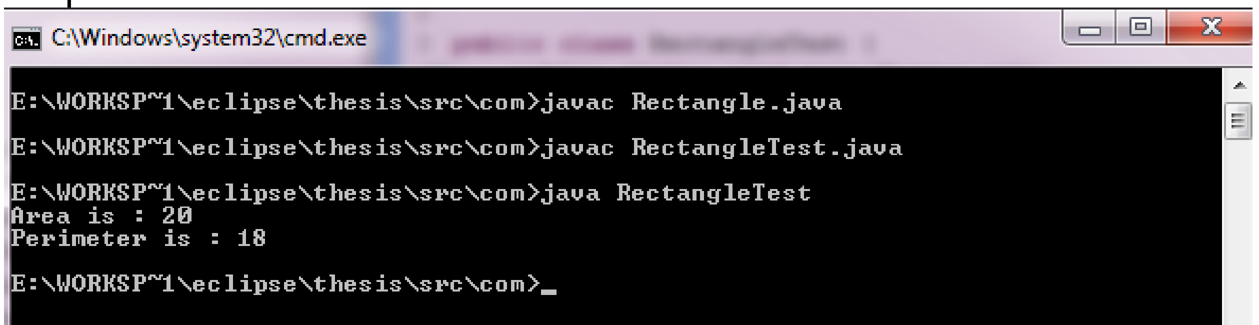
```
public class RectangleTest {
	public static void main(String a[])
	{
		Rectangle rectangle = new Rectangle( 4, 5);
		System.out.println("Area is : "+rectangle.calculateArea());
		System.out.println("Perimeter is :
"+rectangle.calculatePerimeter());
	}
}
```

Solve together:

In this example, we have two classes. The Rectangle and RectangleTest class. As you see, the Rectangle has two instance variables, width and height, and two instance methods, calculateArea and calculatePerimeter. Also has constructor of this class, which receives two int type variables, width and height.

The RectangleTest class just for test or run. It will create an object of Rectangle, gives needed parameters through constructor and calls methods.

Output:



Section 2

Tasks

1. Write a class Triangle, which takes two parameters base and height and calculates area.
2. Write a class RightTriangle, which takes two parameters base and height and calculates hypothesis.
3. Write a class StaticCalc, which takes array of integers and calculates the:
    - getSum() – sum of integers;
    - getCount() – number of integers;
    - getAvarege() - average of ints;
    - getMax() – maximum;
    - getMin() – minimum;

4. Create a class called `Employee` that includes three pieces of information as instance variablesa first name (type `String`), a last name (type `String`) and a monthly salary (`double`). Your class should have a constructor that initializes the three instance variables. Provide a set and a get method for each instance variable. If the monthly salary is not positive, set it to `0.0`. Write a test application named `EmployeeTest` that demonstrates class `Employee`'s capabilities. Create two `Employee` objects and display each object's yearly salary. Then give each `Employee` a 10% raise and display each `Employee`'s yearly salary again.
5. Write a class Student, which takes three parameters, name, surname and university. The student class should have setName(String), change name, setSurname(String), change surname, setUniversity(String), change university, getStudent(), print student info methods.
6. Given n length string, you should put star sign (*) between adjacent letters if there exists. Use recursion.

   For example: “hello” = “hel*lo”
   “aloooo” = “alo*o*o*o”
   “open” = “open”
7. (Recursion) Given n length string, write program, that puts char ‘q’ at the end of given string if there exists. Without loops.

   For example: “hello” = “hello”
   “exit” = “eitx”
   “UNIX” = “UNIX”
   “txxxxxeeext” = “teeetxxxxxx”

CHAPTER NINE

class StringTokenizer. Java I/O. ArrayList

class StringTokenizer

In this section, we study Java's `StringTokenizer` class (from package `java.util`), which breaks a string into its component tokens. Tokens are separated from one another by **delimiters**, typically whitespace characters such as space, tab, newline and carriage return. Other characters can also be used as delimiters to separate tokens.

StringTokenizer(String str, String delim)	constructor
boolean hasMoreTokens()	returns `true` if more tokens exist.
String nextToken()	returns the next token; throws a `NoSuchElementException` if there are no more tokens.

String nextToken(String delim)	returns the next token after switching to the new delimiter set. The new delimiter set is subsequently used.
int countTokens()	returns the number of tokens still in the string.

Java I/O

Java provides the `Reader` and `Writer` `abstract` classes, which are Unicode two-byte, character-based streams. Most of the byte-based streams have corresponding character-based concrete `Reader` or `Writer` classes.

Classes BufferedReader and BufferedWriter enable buffering for character-based streams. Remember that character-based streams use Unicode characterssuch streams can process data in any language that the Unicode character set represents.

Any Reader can be wrapped in a BufferedReader to make it easy to read full lines of text. If reader is of type Reader, then a BufferedReader wrapper can be created for reader with

```
BufferedReader in = new BufferedReader( reader );
```

ArrayLists

In many programming languages—in particular, in C—you have to fix the sizes of all arrays at compile time. Programmers hate this because it forces them into uncomfortable trade-offs. How many employees will be in a department? Surely no more than 100. What if there is a humongous department with 150 employees? Do we want to waste 90 entries for every department with just 10 employees?

In Java, the situation is much better. You can set the size of an array at run time.

```
int arraySize = n;
Employee[] staff = new Employee[arraySize];
```

Of course, this code does not completely solve the problem of dynamically modifying arrays at run time. Once you set the array size, you cannot change it easily. Instead, the easiest way in Java to deal with this common situation is to use another Java class, classed ArrayList. The ArrayList class is similar to an array, but it automatically adjusts its capacity as you add and remove elements, without your needing to write any code. Here we declare and construct an array list that holds `Employee` objects:

```
ArrayList<Employee> staff = new ArrayList<Employee>();
```

Then you can add huge number of Employees as you wish

```
Employee  emp1 = new Employee("Berik Serikov");
staff.add(emp1);
```

ArrayList<T>()	constructs an empty array list.
ArrayList<T>(int initialCapacity)	constructs an empty array list with the specified capacity.
boolean add(T obj)	appends an element at the end of the array list. Always returns `true`.
int size()	returns the number of elements currently stored in the array list. (This is different from, and, of course, never larger than, the array list's capacity.)
void ensureCapacity(int capacity)	ensures that the array list has the capacity to store the given number of elements without relocating its internal storage array.

Random

There is something in the air of a casino that invigorates people from the high rollers at the plush mahogany-and-felt craps tables to the quarter poppers at the one-armed bandits. It is the **element of chance**, the possibility that luck will convert a pocketful of money into a mountain of wealth. The element of chance can be introduced in a program via an object of class **Random** (package java.util) or via the static method random of class Math. Objects of class Random can produce random **boolean, byte, float, double, int, long** and **Gaussian values**, whereas Math method random can produce only double values in the range 0.0 < x < 1.0, where x is the value returned by method random. In the next several examples, we use objects of class Random to produce random values.

A new random-number generator object can be created as follows:

```
Random random = new Random();
     int a = random.nextInt();
```

Examples:

<u>Section 1.</u>

Exercise 1:

Java Code:

```
import java.io.*;
import java.util.*;
public class MixedTest {
      public static void main(String args[]) throws IOException
      {
            BufferedReader reader = new BufferedReader(new FileReader(new
File("path_to_your_file")));
            Random random = new Random();

            String str = "";
            int a = 0;
```

```
            while(( a = reader.read()) != -1)
            {
                  str+=(char)a;
            }

            StringTokenizer tokens = new StringTokenizer(str," ");
            ArrayList<String> stringList = new ArrayList<String>();
            while(tokens.hasMoreTokens())
            {
                  stringList.add(new String(tokens.nextToken()));
            }
            for (int i = 0; i < stringList.size(); i++)
            {
                  System.out.print(stringList.get(i)+" ");
            }
            System.out.println();
            int randomGenerator = random.nextInt(stringList.size());
            System.out.println("String at position "+randomGenerator+ " is '"
+ stringList.get(randomGenerator)+"'");
      }
}
```

Solve together:

In this example we used all topics we passed.

- The program will read data from file by BufferedReader. ;
- Divide into tokens (white spaces);
- Put this tokens into stringList ArrayList (new String());
- Access's all elements of stringListrandomly;

The file may contain any text. Our cases the text in file is “I am 1st year student in SDU” Try to run program more than one. Do you have different output?

Output:

```
C:\Windows\system32\cmd.exe

E:\workspace\eclipse\thesis\src\com>javac MixedTest.java

E:\workspace\eclipse\thesis\src\com>java MixedTest
I am 1st year student in SDU
String at position 4 is 'student'

E:\workspace\eclipse\thesis\src\com>_
```

Section 2.

Tasks

1. Write an application that reads a line of text from the keyboard and prints a table indicating the number of occurrences of each letter of the alphabet in the text. For example, the phrase

```
To be, or not to be: that is the question:
```

contains one "a," two "b's," no "c's," and so on.

2. Write an application that reads a line of text and prints a table indicating the number of occurrences of each different word in the text. For example, the lines

```
To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
```

   contain the word "to" three times, the word "be" two times, the word "or" once, and so on. A more interesting (and useful) printout should then be attempted in which the words are sorted alphabetically.

3. Write an application that reads a line of text and prints a table indicating the number of one-letter words, two-letter words, three-letter words, and so on, appearing in the text. For example "I am from South KZ"

Text Length	Occurance
1	1
2	2
3	0
4	1
5	1
6	0

4. Write an application that reads a date in the first format and prints it in the second format.
   First: 27/08/1990
   Second: August 27, 1990
5. Computers are playing an increasing role in education. Write a program that will help an elementary school student learn multiplication. Use a `Random` object to produce two positive one-digit integers. The program should then prompt the user with a question, such as

```
How much is 6 times 7?
```

   The student then inputs the answer. Next, the program checks the student's answer. If it is correct, display the message `"Very good!"` and ask another multiplication question. If the answer is wrong, display the message `"No. Please try again."` and let the student try the same question repeatedly until the student finally gets it right. A separate method should be used to generate each new question. This method should be called once when the application begins execution and each time the user answers the question correctly.

6. Write a program that reads in a series of first names and stores them in a `StudentList (ArrayList)`. Do not store duplicate names. Allow the user to search for a first name.

CHAPTER TEN

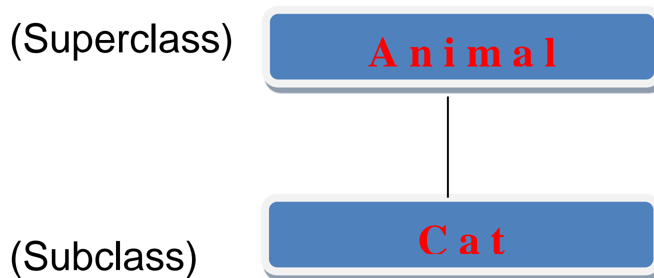
Inheritance. Access Modifiers. static import

This chapter continues our discussion of object-oriented programming (OOP) by introducing one of its primary features inheritance, which is a form of software reuse in which a new class is created by absorbing an existing class's members and embellishing them with new or modified capabilities. With inheritance, programmers save time during program development by reusing

proven and debugged high-quality software. This also increases the likelihood that a system will be implemented effectively.

Superclasses and Subclasses

Inheritance relationships form tree-like hierarchical structures. A superclass exists in a hierarchical relationship with its subclasses. When classes participate in inheritance relationships, they become "affiliated" with other classes. A class becomes either a superclass, supplying members to other classes, or a subclass, inheriting its members from other classes. In some cases, a class is both a superclass and a subclass. For example we have a class Animal and class Cat. The Cat class inherits from Animal class. It means that Cat class inherited all Animal class members. Superclass is Animal, subclass is Cat.



Constructors in Subclasses

As we explained in the preceding section, instantiating a subclass object begins a chain of constructor calls in which the subclass constructor, before performing its own tasks, invokes its direct superclass's constructor either explicitly **(super)** or implicitly calling the superclass's default constructor or no-argument constructor.

Similarly, if the superclass is derived from another class, the superclass constructor invokes the constructor of the next class up in the hierarchy, and so on.

The last constructor called in the chain is always the constructor for class Object. More info in examples section.

public, private and protected *Access Modifiers*

A class's **public** members are accessible wherever the program has a reference to an object of that class or one of its subclasses.

A class's **private** members are accessible only from within the class itself. A superclass's private members are not inherited by its subclasses.

A superclass's **protected** members can be accessed by members of that superclass, by members of its subclasses and by members of other classes in the same package.

Modifier	Class	Package	Subclass	World
public	y	y	y	Y
protected	y	y	Y	n
no modifier	y	y	n	n
private	y	n	n	n

static *Import*

A **static import** declaration enables programmers to refer to imported static members as if they were declared in the class that uses them the class name and a dot (.) are not required to use an imported static member.

A static import declaration has two forms one that imports a particular static member (which is known as **single static import**) and one that imports all static members of a class (which is known as **static import on demand**). The following syntax imports a particular static member:

```
import static packageName.ClassName.staticMemberName;
```

where packageName is the package of the class (e.g., java.lang), ClassName is the name of the class (e.g., Math) and staticMemberName is the name of the static field or method (e.g., PI or abs). The following syntax imports all static members of a class:

```
import static packageName.ClassName.*;
```

where packageName is the package of the class (e.g., java.lang) and ClassName is the name of the class (e.g., Math). The asterisk (*) indicates that all static members of the specified class should be available for use in the class(es) declared in the file. Note that static import declarations import only static class members. Regular import statements should be used to specify the classes used in a program. In Java code:

```
import static java.lang.Math.*;
public class StaticImport
{
     public static void main( String args[] )
     {
        System.out.println(sqrt( 900.0 ) );
        System.out.println(ceil( -9.8 ) );
        System.out.println(log( E ) );
        System.out.println(cos( 0.0 ) );
     }

}
```

Examples:

Section 1.

Exercise 1:

Java Code:

```
// Superclass Animal
public class Animal extends Object {
      private String name;

      public Animal(String name)
      {
            this.name = name;

            System.out.println("Animal can walk");
      }
      public String getName()
      {
            return name;
      }
}
//Subclass Dog

public class Dog extends Animal {

      private String hairColor;
      public Dog(String name, String  hairColor)
      {
            super(name);
            this.hairColor = hairColor;

            System.out.println("Dog can bark");
      }

      public String getHairColor()
      {
            return hairColor;
      }

}
// Test

public class Test {
      public static void main(String[] args) {
            Dog dog = new Dog("Boribasar","BLACK");
            System.out.println(dog.getName());
            System.out.println(dog.getHairColor());
      }

}
```

Solve together:

In this case, class Dog inherited all fields from Animal class. The keyword **super** is for calling upper classes constructor. In output you can see order of calling constructors.

Also we have two duplicate variable names ***name*** and ***hairColor*** in class. To differentiate them, we use keyword **this**. The keyword **this** means - current class's variable.

We created subclasses object "new Dog()". Where did dog class get "getName()" method? Because of, inheritance. Dog class inherited all accessible members from superclass.

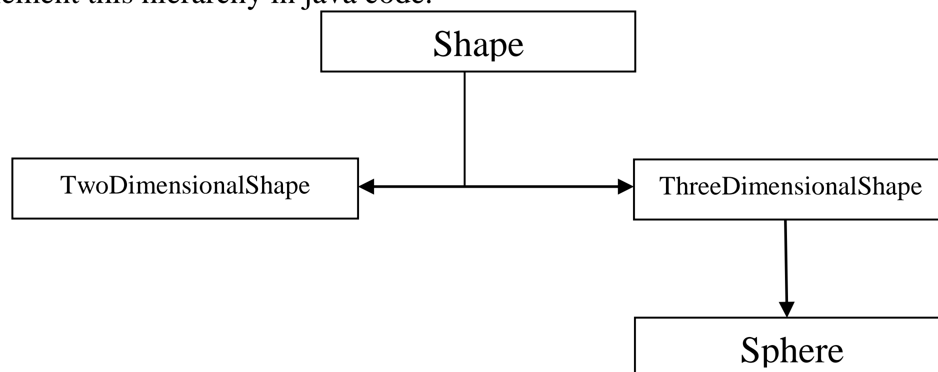
Output:

```
Administrator: C:\Windows\system32\cmd.exe

E:\workspace\eclipse\ExtraLesson\src\inheritance>javac Animal.java Dog.java Test
.java

E:\workspace\eclipse\ExtraLesson\src\inheritance>java Test
Animal can walk
Dog can bark
Name is : Boribasar
Color is : BLACK

E:\workspace\eclipse\ExtraLesson\src\inheritance>_
```

Section 2.

Tasks

1. Write a class Shape. Triangle class has variable **name** and method **getName**. Write subclass RightTriangle. RightTriangle class has variables **name, base** and **height** and method calculateArea**.** You should calculate area of Right Triangle. Write Test class to test them.
2. Modify previous task, the variable **name** *protected* now and move method **getName** to RightTrianle class. Run it. Do you have difference?
3. Implement this hierarchy in java code.



4. Write an inheritance hierarchy for classes Quadrilateral, TRapezoid, Parallelogram, Rectangle and Square. Use Quadrilateral as the superclass of the hierarchy. Make the hierarchy. Specify the instance variables and methods for each class. The private instance variables of Quadrilateral should be the x-y coordinate pairs for the four endpoints of the Quadrilateral. Write a program that instantiates objects of your classes and outputs each object's area (except Quadrilateral).
5. Draw an inheritance hierarchy for students at a university. Use Student as the superclass of the hierarchy, then extend Student with classes UndergraduateStudent and GraduateStudent. Continue to extend the hierarchy as deep as possible. For example, Freshman, Sophomore, Junior and Senior might extend UndergraduateStudent, and DoctoralStudent and MastersStudent might be subclasses of GraduateStudent. Afterdrawing the hierarchy, discuss the relationships that exist between the classes.

CHAPTER ELEVEN

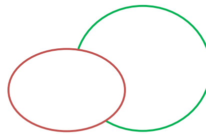
Polymorphism, Abstract classes, Keyword *final*

We now continue our study of object-oriented programming by explaining and demonstrating **polymorphism** with inheritance hierarchies. Polymorphism enables us to "program in the general" rather than "program in the specific." In particular, polymorphism enables us to write programs that process objects that share the same superclass in a class hierarchy as if they are all objects of the superclass.

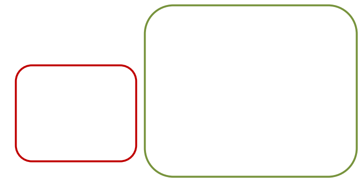
Polymorphism

Consider a program that deals with shapes drawn on the screen. Let's say that the shapes include rectangles, ovals, and roundrects of various colors. (A "roundrect" is just a rectangle with rounded corners.) Three classes, Rectangle, Oval, and RoundRect, could be used to represent the three types of shapes. These three classes would have a common superclass, Shape, to represent features that all three shapes have in common. The Shape class could include instance variables to represent the color, position, and size of a shape, and it could include instance methods for changing the color, position, and size. Changing the color, for example, might involve changing the value of an instance variable, and then redrawing the shape in its new color:

Rectangles



Ovals



RoundRects

```
class Shape {
      Color color;
      void setColor(Color newColor) {
            color = newColor;
            redraw();
      }
      void redraw() {
           . . . . .
      }
}
```

Now, you might see a problem here with the method redraw(). The problem is that each different type of shape is drawn differently. The method setColor() can be called for any type of shape. How does the computer know which shape to draw when it executes the redraw()? Informally, we can answer the question like this: The computer executes redraw() by asking the shape to **redraw itself**.

Every shape object knows what it has to do to redraw itself. In practice, this means that each of the specific shape classes has its own redraw() method:

```
class Rectangle extends Shape {
      void redraw() {
            . . .
      }
      . . .
}
class Oval extends Shape {
      void redraw() {
            . . .
      }
```

```
        . . .
}
```

We say that the **redraw()** method is polymorphic. A method is polymorphic if the action performed by the method depends on the actual type of the object to which the method is applied. Polymorphism is one of the major distinguishing features of object-oriented programming. Perhaps this becomes more understandable if we change our terminology a bit: In object-oriented programming, calling a method is often referred to as sending a message to an object.

Abstract classes

Whenever a Rectangle, Oval, or RoundRect object has to draw itself, it is the redraw()method in the appropriate class that is executed. This leaves open the question, What does the redraw() method in the Shape class do? How should it be defined?
The answer may be surprising: We should leave it blank! The fact is that the class Shape represents the abstract idea of a shape, and there is no way to draw such a thing. Only particular, concrete shapes like rectangles and ovals can be drawn.

Nevertheless the version of redraw() in the Shape class itself will never actually be called. In fact, if you think about it, there can never be any reason to construct an actual object of type Shape! You can have variables of type Shape, but the objects they refer to will always belong to one of the subclasses of Shape. We say that Shape is an **abstract** class. An abstract class is one that is not used to construct objects, but only as a basis for making subclasses. An abstract class exists only to express the common properties of all its subclasses. A class that is not abstract is said to be concrete. You can create objects belonging to a concrete class, but not to an abstract class. A variable whose type is given by an abstract class can only refer to objects that belong to concrete subclasses of the abstract class.

Similarly, we say that the redraw() method in class Shape is an **abstract** method, since it is never meant to be called. In fact, there is nothing for it to do—any actual redrawing is done by redraw() methods in the subclasses of Shape. The redraw() method in Shape has to be there. But it is there only to tell the computer that all Shapes understand the redraw message. As an abstract method, it exists merely to specify the common interface of all the actual, concrete versions of redraw() in the subclasses of Shape. There is no reason for the abstract redraw() in class Shape to contain any code at all.

Shape and its redraw() method are semantically abstract. You can also tell the computer, syntactically, that they are abstract by adding the modifier "abstract" to their definitions.
For an abstract method, the block of code that gives the implementation of an ordinary method is replaced by a semicolon. An implementation must be provided for the abstract method in any concrete subclass of the abstract class. Here's what the Shape class would look like as an abstract class:

```
public abstract class Shape {
      Color color;
       void setColor(Color newColor) {
            color = newColor;
            redraw();
      }
      abstract void redraw();
      . . .
}
```

Once you have declared the class to be abstract, it becomes illegal to try to create actual objects of type Shape, and the computer will report a syntax error if you try to do so.

final Methods and Classes

We saw that variables can be declared final to indicate that they cannot be **modified** after they are declared and that they must be initialized when they are

declaredsuch variables represent constant values. It is also possible to declare methods and classes with the final modifier.

A method that is declared final in a superclass cannot be **overridden** in a subclass. Methods that are declared **private** are implicitly final, because it is impossible to override them in a subclass. Methods that are declared **static** are also implicitly final, because static methods cannot be overridden either. A final method's declaration can never change, so all subclasses use the same method implementation and calls to final methods are resolved at compile timethis is known as **static binding**.

A class that is declared `final` cannot be a superclass (i.e., a class cannot extend a `final` class). All methods in a `final` class are implicitly `final`.

this and *super*

Keyword this, is to refer to "this object," the one right here that this very method is in. If x is an instance variable in the same object, then this.x can be used as a full name for that variable. If otherMethod() is an instance method in the same object, then this.otherMethod() could be used to call that method.

Java also defines another special variable, named super, for use in the definitions of instance methods. The keyword **super** is for use in a subclass.

Let's say that the class that you are writing contains an instance method named doSomething(). Consider the subroutine call statement super.doSomething(). Now, super doesn't know anything about the doSomething() method in the subclass. It only knows about things in the superclass, so it tries to execute a method named doSomething() from the superclass.

Examples:

Section 1.

Exercise 1:

Java Code:

// Abstract class Shape

```
public abstract class Shape {
	private String color = null;
	public Shape(String color)
	{
		this.color = color;
	}
	public void setColor(String newColor)
	{
		this.color = newColor;
	}
	public String getColor()
	{
		return color;
	}
	public abstract void draw();
}
```

// Derived class Recangle

```
public class Rectangle extends Shape{
	private int x,y;
	public Rectangle(String color, int x, int y)
	{
		super(color);
		this.x = x;
```

```
		this.y = y;
	}

	public void draw() {
		System.out.println("I am "+getColor()+" colored  in "+x+","+y+" position");
	}
}
```

// Derived class Oval

```
public class Oval extends Shape {
	private String color;
	private int x,y;
	public Oval(String color, int x, int y)
	{
		super(color);
		this.x = x;
		this.y = y;
	}
	public void draw() {
		System.out.println("I am "+getColor()+" colored in "+x+","+y+" position");

	}
}
```

//Tester class

```
public class Tester {
	public static void main(String args[])
	{
		Rectangle rectangle = new Rectangle("RED", 10, 20);
		Oval oval = new Oval("BLUE", 5, 100);
		rectangle.draw();
		oval.draw();
	}
}
```

Solve together:

Firstly we created abstract class **Shape** with abstract method **draw**. And two **Rectangle** and **Oval** classes extends from **Shape**. It means that, two classes **must** implement **draw** method. When we call draw method, each class will use own draw methods. Because of, abstract. But getColor method from class Shape. Because of, not abstract.

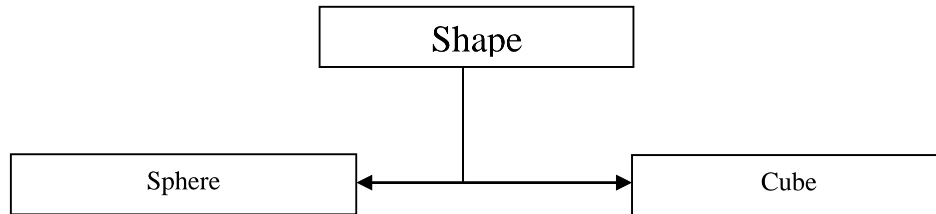
Output:

```
Administrator: C:\Windows\system32\cmd.exe

E:\workspace\eclipse\thesis\src\polymorphism>javac Shape.java Rectangle.java Ova
l.java Tester.java

E:\workspace\eclipse\thesis\src\polymorphism>java Tester
I am RED colored  in 10,20 position
I am BLUE colored in 5,100 position

E:\workspace\eclipse\thesis\src\polymorphism>_
```

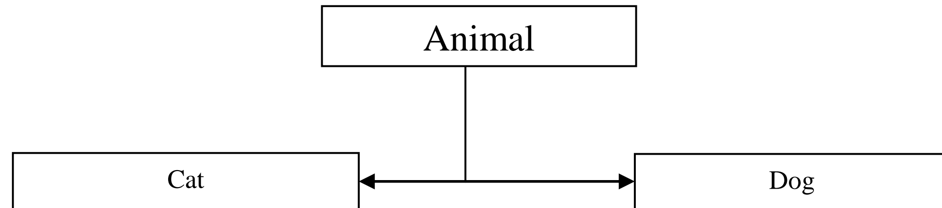
Section 2.

Tasks

1. As you see in the followed figure, class Shape has abstract method **calculateArea**. Sphere and Cube must override this method, because each classes calculates area differently. Sphere and Cube have getWidth, getHeight and etc. methods.



2. Implement this followed figure in java code. Class Animal has abstract method move. Dog and Cat must implement move method, because each animal moving differently.



3. In all versions of the class PairOfDice the instance variables die1 and die2 are declared to be public. They really should be private, so that they are protected from being changed from outside the class. Write another version of the PairOfDice class in which the instance variables die1 and die2 are private. Your class will need "getter" methods that can be used to find out the values of die1 and die2. (The idea is to protect their values from being changed from outside the class, but still to allow the values to be read.) Include other improvements in the class, if you can think of any. Test your class with a short program that counts how many times a pair of dice is rolled, before the total of the two dice is equal to two.
4. Given a string, return a string made of the chars at indexes 0,1, 4,5, 8,9 ... so "kittens" yields "kien".

   altPairs("kitten") → "kien"
   altPairs("Chocolate") → "Chole"
   altPairs("CodingHorror") → "Congrr"
5. Given a string, return a version where all the "x" have been removed. Except an "x" at the very start or end should not be removed.

   stringX("xxHxix") → "xHix"
   stringX("abxxxcd") → "abcd"
   stringX("xabxxxcdx") → "xabcdx"

CHAPTER TWELVE

Exception. Interface. Package

In a perfect world, users would never enter data in the wrong form, files they choose to open would always exist, and code would never have bugs. So far, we have mostly presented code as though we lived in this kind of perfect world. It is now time to turn to the mechanisms the Java programming language has for dealing with the real world of bad data and buggy code.

Exception

Suppose an error occurs while a Java program is running. The error might be caused by a file containing wrong information, a flaky network connection, or (we hate to mention it) use of an invalid array index or an attempt to use an object reference that hasn't yet been assigned to an object. Java has many types of exception like:

- RuntimeException
- ArrayIndexOutOfBoundsException
- NullPointerException
- FileNotFoundException
- FileFormatException
- …

This type of exceptions may occur when we are writing our own applications.

Catching Exceptions:

If an exception occurs that is not caught anywhere, the program will terminate and print a message to the console, giving the type of the exception and a stack trace. Graphics programs (both applets and applications) catch exceptions, print stack trace messages, and then go back to the user interface processing loop.

To catch an exception, you set up a `try/catch` block. The simplest form of the `try` block is as follows:

```
try
{
  code
  more code
  more code
}
catch (ExceptionType e)
{
    what must doo..

}
```

The finally Clause

When your code throws an exception, it stops processing the remaining code in your method and exits the method. This is a problem if the method has acquired some local resource that only it knows about and if that resource must be cleaned up. One solution is to catch and rethrow all exceptions. But this solution is tedious because you need to clean up the resource allocation in two places, in the normal code and in the exception code.

Java has a better solution, the finally clause. Here we show you how to properly dispose of a Graphics object. If you do any database programming in Java, you will need to use the same techniques to close connections to the database. The **finally** formula looks like:

```
Graphics g = image.getGraphics();
TRy
{
  // 1
  code that might throw exceptions
  // 2
}
catch (IOException e)
{
  // 3
  show error dialog
  // 4
}
finally
{
  // 5
  g.dispose();
}
```

Declaring New Exception

A new exception class must extend an existing exception class to ensure that the class can be used with the exception-handling mechanism. Like any other class, an exception class can contains fields and methods.

However, a typical new exception class contains only two constructors one that takes no arguments and passes a default exception message to the superclass constructor, and one that receives a customized exception message as a string and passes it to the superclass constructor.

Interface

Interfaces define and standardize the ways in which things such as people and systems can interact with one another. For example, the controls on a radio serve as an interface between **radio users** and a **radio's internal components**. The controls allow users to perform only a limited set of operations (e.g., changing the station, adjusting the volume, choosing between AM and FM), and different radios may implement the controls in different ways (e.g., using push buttons, dials, voice commands).

The *interface* specifies what operations a radio must permit users to perform but **does not** specify how the operations are performed.

Similarly, the interface between a driver and a car with a manual transmission includes the steering wheel, the gear shift, the clutch pedal, the gas pedal and the brake pedal. This same interface is found in nearly all manual transmission cars, enabling someone who knows how to drive one particular manual transmission car to drive just about any manual transmission car.

The components of each individual car may look different, but the components' general purpose is the same to allow people to drive the car.

An **interface declaration** begins with the keyword **`interface`** and contains only constants and `abstract` methods.

To use an interface, a concrete class must specify that it **`implements`** the interface and must declare each method in the interface with the signature specified in the interface declaration.

A class that does not implement all the methods of the interface is an abstract class and must be declared `abstract`.

Packages

As we have seen, Java contains many predefined classes that are grouped into categories of related classes called packages. Together, we refer to these packages as the Java Application Programming Interface (Java API), or the Java class library.

Throughout the text, import declarations specify the classes required to compile a Java program. For example, a program includes the declaration

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

to specify that the program uses class Scanner from the **java.util** package. This allows programmers to use the simple class name Scanner, rather than the fully qualified class name java.util.Scanner, in the code. There you can see some Java API packages:

Package	Description
`java.applet`	The Java Applet Package contains a class and several interfaces required to create Java applets programs that execute in Web browsers.
`java.io`	The Java Input/Output Package contains classes and interfaces that enable programs to input and output data.
`javax.swing`	The Java Swing GUI Components Package contains classes and interfaces for Java's Swing GUI components that provide support for portable GUIs.

Examples:

<u>Section 1.</u>

Exercise 1:

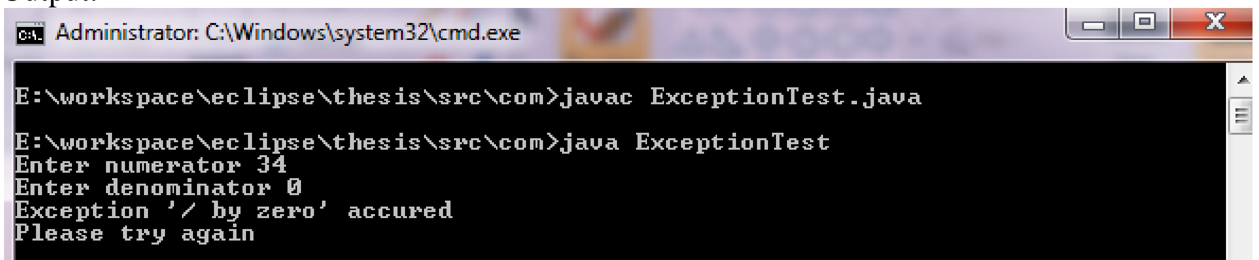
Java Code:

```
import java.util.Scanner;
public class ExceptionTest
{
	public static void main(String[] args)
	{
		Scanner in = new Scanner(System.in);
		int numerator, denominator;
		System.out.print("Enter numerator ");
		numerator = in.nextInt();
		System.out.print("Enter denominator ");
		denominator = in.nextInt();

		try
		{
			System.out.println("Result is: "+(numerator/denominator));
		}catch(ArithmeticException exception)
		{
			System.out.println("Exception '"+exception.getMessage()+"'
accured");
		} finally
		{
			System.out.println("Please try again");
		}

	}
}
```

Solve together:

Here you can see an Arithmetic exception. In math you cannot divide any number by zero. We handled exception and tell the user. The finally clause always works.

Output:



Exercise 2:

Java Code:

// Interface Moveable class

```
public interface Moveable {
	void move(int x, int y);
}

// Class Dog
public class Dog implements Moveable {
	@Override
	public void move(int x, int y) {
		System.out.println("Hi, I'm Dog and I in ("+x+","+y+") position");
	}
```

```
}
//Class Cat
public class Cat implements Moveable {
	@Override
	public void move(int x, int y) {
		System.out.println("Hi, I'm Cat and I in ("+x+","+y+") position");
	}
}
```

Solve together:

Here we created an interface class Moveable and the two classes implement its move method.

Output:

```
Administrator: C:\Windows\system32\cmd.exe

E:\workspace\eclipse\thesis\src\interfaces>javac Moveable.java Dog.java Cat.java
 InterfaceTest.java

E:\workspace\eclipse\thesis\src\interfaces>java InterfaceTest
Hi, I'm Dog and I in (23,14) position
Hi, I'm Cat and I in (67,43) position

E:\workspace\eclipse\thesis\src\interfaces>
```

Section 2.

Tasks

1. Write a program that catches ArrayOutOfIndex exception. You should declare an array and run through this array with loop. Catch exception when index exceeds the length of the array.
2. Write a program that shows a constructor passing information about constructor failure to an exception handler. Define class SomeException, which throws an Exception in the constructor. You program should try to create an object of type SomeException, and catch the exception that is thrown from the constructor.
3. Write a program that illustrates rethrowing an exception. Define methods someMethod and someMethod2. Method someMethod2 should initially throw an exception. Method someMethod should call someMethod2, catch the exception and rethrow it. Call someMethod from method main, and catch the rethrown exception. Print the stack trace of this exception.
4. Write a program that handles MineFileNotFoundException. Create your own MineFileNotFound exception. Catch exception if file doesn't exists.
5. Write an interface class MyInterface.java which contains method sayHello(). Every implemented classes form this class should print out a word "Hello I am from interface".

6. Suppose the string "yak" is unlucky. Given a string, return a version where all the "yak" are removed, but the "a" can be any char. The "yak" strings will not overlap.

    stringYak("yakpak") → "pak"
    stringYak("pakyak") → "pak"
    stringYak("yak123ya") → "123ya"