Primitive Types and Some Simple IO in Java

1. Overview

The learning objectives of this laboratory session are:

- How to build and run a java standalone program without an IDE
- The elements of good programming style
- Understand and get hands on experience with primitive types and wrapper classes
- How to use some simple I/O statements

2. Compiling a Java Program

Turning a Java source program into an object program takes a couple of steps. Assuming that you are using the popular, but awkward, JDK (Java Development Kit) by Sun Microsystems, do the following:

1. **Create the source program with a text editor** (e.g., jEdit, TextPad, ...). Save it in a file with the same name as the public class adding the extension ".java" (e.g., Greeting.java). A common error is to use a different name for the file and the class. The name before the "." must be the same as the class name, including upper- or lowercase. Many programmers save their source file every 10 minutes or so -- it's quick and saves the aggravation of having to type it again if there is a system crash.
2. Open a DOS command window and **cd** to the directory containing the source file. This is easy if you've used short directory names without spaces.
3. **Compile** the source program (**Greeting.java** in this example) with the following DOS command:

   ```
   javac Greeting.java
   ```

   This produces one or more ".class" files, which are the object (Java byte code) form of Java programs. You can produce a ".exe" file from this, but that isn't normally done.

4. **Run** it with:

   ```
   java Greeting
   ```

   This loads the **Greeting.class** file and all necessary classes. Execution starts with the main method in the Greeting class. Continue in this cycle until the program works.

Note. Java sources have the extension **.java**. Compiled Java code has the extension **.class**.

2.1. Where Java finds programs

A common way to compile and run a simple Java program is with commands like the following.

```
javac MyProgram.java
java MyProgram
```

Or you may use an IDE that permits you to compile and run. In any case, Java must know, both for compilation and execution of the programs, where to find any library classes that are used. Java (at least after version 1) knows how to find it's own classes, but you have to tell it where to find other libraries you may be using. The standard Microsoft Windows directories are not searched.

You may need to specify a list of directories or jar files where programs can be found by setting the *environment variable* `CLASSPATH`.

2.2. Organizing your work

Whenever you start a **new project**, create a **new directory** for the source files. The directory name should be lowercase letters, with no blanks or other punctuation.
Multiple classes of larger programs are usually grouped together into *packages*. Following the optional *package* declaration, you can have *import* statements, which allow you to specify classes that can be referenced without qualifying them with their package.
Packages are directories / folders that contain the Java classes, and are a way of grouping related classes together. For small programs it's common to omit a package specification (Java creates what it calls a *default* package in this case).

One class per file

Put each class in its own separate source file. Each source file must be named *exactly* the same as the class, plus a ".java" suffix. For example, if the class is named "Test", the file must be named "Test.java" (and not "test.java").
It's possible to put more than one class in a file and have everything work. But this doesn't scale up as you create larger programs and development tools. IDEs (like NetBeans) require each class to be in a separate source file. Also, a very common development tool, Ant, works best when each class is in its own source file.

3. Programming style

The following is a based on excerpts from the article "Good Java Style" by Thornton Rose.
Several reasons for using good style [from "Java Code Conventions," Sun Microsystems]:

- 80% of the lifetime cost of a software product goes to maintenance.
- Hardly any software is maintained for its whole life by the original author(s).
- Using good style improves the maintainability of software code.
- If the source code is shipped with the software, it should be as well-packaged, clean, and professional as the rest of the product.

Writing code with good style also provides the following benefits:

- It improves the readability, consistency, and homogeneity of the code, which makes it easier to understand and maintain.
- It makes the code easier to trace and debug, because it's clear and consistent.
- It allows you to continue more easily where you or another programmer stopped, particularly after a long period of time.
- It increases the benefit of code walkthroughs, because the participants can focus more on what the code is doing.

3.1. General Guidelines

Writing Java with good style is not hard, but it does require attention to detail. Here are some general guidelines to follow:

- Make the code clear and easy to read.
- Make the code consistent.

- Use obvious identifier names.
- Logically organize your files and classes.
- Have only one class per file (not including inner classes).
- Use a maximum line width of 80-90 characters.
- Use whitespace and/or other separators judiciously.
- Use spaces instead of tabs for indentation (change of tab size will not change the aspect of your code, then) .

Braces and Indentation

Indent style or the placement of braces ("{" and "}") and the associated indentation of code, is another of the religious issues related to writing code. There are several indent styles common to C-style languages like Java. Many favor K&R style with opening brace in the same line with the statement containing a block and closing brace at the same indentation level of indentation as the statement which contains the block. Comment style is also a part of the programming style.

3.2. Comments in Java

Computer programs are read by both computes and humans. You write Java instructions to tell the computer what to do. You must also write comments to explain to humans what the program does. Of course, Java can't understand them because they are written in various languages.
Java ignores all comments. There is, however, a program called **javadoc** which reads certain kinds of comments and produces HTML documentation
Use spaces and blank lines in your programs. One of the most effective ways to make a program readable is to put spaces in at key points. There are several styles for doing this. Even more important is to put blank lines in your program. These should separate sections of code. There should be a blank line between each group of statements that belong together logically.
There are several kinds of comments:

// comments – single line

After the two **//** characters, Java ignores everything to the end of the line. This is the most common type of comment.

```
//--- local variables ---
int nItems;  // number of items.
int score;   // count of number correct minus number wrong.
```

/* ... */ comments -- multiple lines

After the **/*** characters, Java will ignore everything until it finds a ***/**. This kind of comment which you know form C, can cross many lines, and is commonly used to "comment out" sections of code -- making Java code into a comment while debugging a program. For example,

```
/* Use comments to describe variables or sections of the program.
   They are very helpful to everyone who reads your programs:
   First, YOURSELF, then teacher, your boss, etc, but especially yourself!
*/
```

javadoc comments

Comments that start with **/**** are used by the **javadoc** program to produce HTML documentation for the program. The Java documentation from Sun Microsystems is produced using **javadoc**. It is essential to use this kind of comment for large programs. We strongly advise to use this kind of comments to promote reuse of your code

Best practices with comments:

- Don't write comments to document obvious statements. Assume the reader knows Java.

- Every comment has the potential to create an inconsistency between what the comment says, and what the code does. One cause of problems with software is that code is changed over time, but comments are not updated. To avoid this, keep comments next to the code that is documented so that they may be more easily synchronized.

3.3. Identifier Names

Getting the names of things right is extremely important.

Legal Characters

Every name is made from the following characters, starting with a letter:

- Letters: a-z, A-Z, and other alphabetic characters from other languages.
- Digits: 0-9
- Special: _ (underscore)

No names can be the same as a Java keyword. Java keywords are:

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

Examples

apple	This is a legal name. **Lowercase** implies it's a **variable** or **method**.
Apple	This is a different legal name. **Uppercase** implies it's a **class** or **interface**.
APPLE	Yet a different legal name. **All uppercase** implies it's a **constant**.
topleft	Legal, but multiple words should be camelcase.
top_left	Better, but camelcase is preferred to _ in Java.
topLeft	Good Java style
top left	ILLEGAL - no blanks in a name
import	ILLEGAL - same as the Java keyword

Using Uppercase, Lowercase, and "Camelcase" Letters

The conventions for the use of upper- and lowercase is not enforced by compilers, but it is so widely observed, that it should have been. *Camelcase* is the practice of capitalizing the first letter of successive words in multi-word identifiers. Camelcase is much preferred in the Java community over the use of underscores to separate words, or even worse, no distinction made at word boundaries.

Class and interface names - Start with uppercase

Class and interface names start with an uppercase letter, and continue in lowercase. For multiple words, use *camelcase*. Eg, Direction, LogicalLayout, DebugGapSpacer.

Variable and method names - Lowercase

Lowercase is used for variable and method names. If a name has multiple words, use camelcase. Eg, top, width, topLeft, roomWidth, incomeAfterTaxes.

Constants - All uppercase, use _ to separate words

The names of constants (typically declared *static final*) should be in all uppercase. For example, BorderLayout.NORTH. When constant names are made from multiple words, use an underscore to separate words, eg, JFrame.EXIT_ON_CLOSE

Readable names are more important than most comments

Java doesn't care if your names are readable, but it's really important to make your names readable to humans.

4. Java Primitive Types

4.1. Numbers

There are two general kinds of numbers in Java and most other programming languages: binary **integers** and binary **floating-point** numbers (sometimes called *real* numbers). Although these numbers are stored in the computer as binary numbers, you will usually use decimal numbers in your Java source program, and the Java compiler will translate them to the correct binary form.

Integers

The are four types of integers in Java: byte, short, int, long. **The most common is int**. All integers are stored in **signed, two's-complement**, format.
Technically, `char` is an unsigned integer type although it is almost exclusively used to store characters. Making it integer is largely because of Java's legacy from C++. Don't use `char` for integers unless you are sure of what you're doing.

Classes. In addition to the primitive types, there are two classes used for integers.

- **Integer** - Primarily useful for utility methods and to put in the **Collections** data structure classes.
- **BigInteger** - Used where unbounded arithmetic is important.

Java stores all integers in memory as binary numbers.

Type	**Size**		**Range**	
name	*bytes*	*bits*	*minimum*	*maximum*
byte	1	8	-128	+127
short	2	16	-32,768	+32,767
int	4	32	-2,147,483,648	+2,147,483,647
long	8	64	-9,223,372,036,854,775,808	+9,223,372,036,854,775,807

Here is how to write **decimal integer** literals (constants).

- `int` literals are written in the usual decimal notation, like 34 or -222.
- `long` literals are written by adding an L (or lowercase l although this is almost impossible to distinguish from the digit 1), eg, 34L or -222L.

There is no way to write a literal `byte` or `short`, although sometimes Java will automatically cast an **int** literal to the appropriate type.

Hexadecimal literals. You can write an int in hexadecimal by prefixing the hexadecimal number with the digit zero followed by the letter x, "0x" or "0X". The hexadecimal digits are 0-9 and the letters a-f in upper- or lowercase.

```
int i;
i = 0x2A;  // assigns decimal 42 to i.
```

Operations may produce numbers which are too large (overflow) to be stored in an int. No error is caused in this case; the result is simply an incorrect number. Division by zero will cause an execution exception (ArithmeticException). Use BigInteger to prevent arithmetic overflow.

Floating-point

Floating-point numbers are like *real* numbers in mathematics, for example, 3.14159, -0.000001. Java has two kinds of floating-point numbers: `float` and `double`, both stored in IEEE-754 format. The **default type** when you write a floating-point literal is **`double`**.

Type	Size		Range	Precision
name	*bytes*	*bits*	*approximate*	*in decimal digits*
`float`	4	32	+/- 3.4 * 10^{38}	6-7
`double`	8	64	+/- 1.8 * 10^{308}	15

Because there are only a limited number of bits in each floating-point type, some numbers are inexact, just as the decimal system can not represent some numbers exactly, for example 1/3. The most troublesome of these is that 1/10 can not be represented exactly in binary.

Floating-point literals
There are two types of notation for floating-point numbers. Any of these numbers can be followed by "F" (or "f") to make it a `float` instead of the default `double`.
Standard (American) notation which is a series of digits for the integer part followed by a decimal point followed by a series of digits for the fraction part. Eg, 3.14159 is a `double`. A sign (+ or -) may precede the number.
Scientific notation which is a standard floating-point literal followed by the letter "E" (or "e") followed by an optionally signed exponent of 10 which is used as a multiplier (ie, how to shift the decimal point). Generally scientific notation is used only for very large or small numbers.

Scientific	Standard
1.2345e5	123450.0
1.2345e+5	123450.0
1.2345e-5	0.000012345

Infinity and NaN
No exceptions are generated by floating-point operations. Instead of an interruption in execution, the result of an operation may be positive infinity, negative infinity, or NaN (not a number). Division by zero or overflow produce infinity. Subtracting two infinities produces a NaN. Use methods in the wrapper classes (Float or Double) to test for these values.

4.2. Converting Strings to Numbers

To convert a string value to a number (for example, to convert the String value in a text field to an int), use these methods. Assume the following declarations:

String s;
int i;
long l;
float f;
double d;

type	*Example statement*
`int`	`i = Integer.parseInt(s);`
`long`	`l = Long.parseLong(s);`

`float`	`f = Float.parseFloat(s);`
`double`	`d = Double.parseDouble(s);`

If *s* is null or not a valid representation of a number of that type, these methods will *throw* (generate) a `NumberFormatException`.

Handling NumberFormatExceptions

Put number conversions inside a **`try . . . catch`** statement so that you can do something if bad input is entered. The conversion method will *throw* a NumberFormatException when there is bad input. *Catch* the NumberFormatException, and do something to handle this error condition. Put your conversion in the `try` clause, and the error handling in the `catch` clause. Here is an example of the kind of utility function you might write to do this checking.

```
//--- Utility function to get int using a dialog.
public static int getInt(String mess) {
    int val;
    while (true) { // loop until we get a valid int
        String s = JOptionPane.showInputDialog(null, mess);
        try
        {
            val = Integer.parseInt(s);
            break;  // exit loop with valid int >>>>>>>>>>>>>>>>>>>>>>
        }
        catch (NumberFormatException nx)
        {
            JOptionPane.showMessageDialog(null, "Enter valid integer");
        }
    }
    return val;
}//end getInt
```

Non-decimal Integers

Convert integers with some base (radix) other than 10 by using these two methods. Typically these will be hexadecimal (base 16) or binary (base 2) numbers.

type	*Example statement*
`int`	`i = Integer.parseInt(s, radix);`
`long`	`l = Long.parseLong(s, radix);`

For example, to convert a string containing the hexadecimal number "F7" to an integer, call
`i = Integer.parseInt("F7", 16)`

Java also provides classes for arbitrary precision arithmetic on decimals: `BigDecimal`.

4.3. Boolean

The primitive type `boolean` has only two possible values: `true` and `false`.
The two values are written with the reserved words `true` and `false`.
The `if`, `for`, `while`, and `do` statements all require boolean values. Usually these are written as boolean valued expressions, using operators which produce boolean values.

Comparison operators

Comparison operators are used to compare two primitive values (rarely objects).

Op	Name	Meaning
`i < j`	less than	`6 < 24` is true.
`i <= j`	less than or equal	`6 <= 24` is true.
`i == j`	equal	`6 == 24` is false.
`i >= j`	greater than or equal	`10 >= 10` is true.
`i > j`	greater than	`10 > 10` is false.
`i != j`	not equal	`6 != 24` is true.

Logical operators

Op	Name	Meaning
`a && b`	and	The result is true only if both *a* and *b* are true.
`a || b`	or	The result is true if either *a* or *b* is true.
`!a`	not	true if a is false and false if a is true.

Other operators and methods returning boolean values
The `instanceof` operator.
Many **methods** return boolean values, eg, `equals`, and methods that begin with "is". If you are writing your own boolean method, starting the name with "is" is a good practice.
Less common **logical operators**: &, |, and ^ with boolean operands. These are generally used with bits. || (or) and && (and) are preferred to | and & because they are *short-circuit* operators that can stop the evaluation when one of the operands determines the resulting value.

Boolean variables

You can declare boolean variables and test them. For example, this simple bubble sort keeps looping until there were no exchanges, which means that everything must be sorted. This is only an example, not a good way to sort.

```
void bubbleSort(int[] x, int n) {
    boolean anotherPass;  // true if something was out of order
    do {
        anotherPass = false;  // assume everything sorted
        for (int i=0; i<n-1; i++) {
            if (x[i] > x[i+1]) {
                int temp = x[i]; x[i] = x[i+1]; x[i+1] = temp; // exchange
                anotherPass = true;  // something wasn't sorted, keep going
            }
        }
    } while (anotherPass);
}
```

4.4. Character

Character class static methods

Character Class Methods
Character class is used mostly for static methods to test char values.

b =	Character.isDigit(c)	true if c is digit character.
b =	Character.isLetter(c)	true if c is letter character.
b =	Character.isLetterOrDigit(c)	true if c is letter or digit.
b =	Character.isLowerCase(c)	true if c is lowercase char.

b =	Character.isUpperCase(c)	true if c is uppercase char.
b =	Character.isWhitespace(c)	true if c is space, tab, ...
c =	Character.toLowerCase(c)	Lowercase version of c.
c =	Character.toUpperCase(c)	Uppercase version of c.

ANSI/ASCII and Extended Latin Sections of Unicode

Unicode attempts to represent the characters in all current human languages, as well as numerous special symbols. The most common implementation of it uses 16 bits, which is 65,536 characters (many are not yet assigned a graphic). The first 128 codes are identical to ANSI/ASCII (American National Standards Institute / American Standard Code for Information Interchange). Of the ASCII codes, the first 32 are control codes. The first 256 codes are the same as ISO-8859-1 (Latin-1), which includes ASCII of course. Below is a table which shows this common first part of the Unicode character set.

	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14	+15
32		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
128	€	•	‚	ƒ	„	…	†	‡	ˆ	‰	Š	‹	Œ	•	Ž	•
144	•	‘	’	“	”	•	–	—	˜	™	š	›	œ	•	ž	Ÿ
160		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯
176	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
208	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
224	à	á	â	Ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
240	ð	ñ	ò	Ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

5. Some simple IO

Java 5's *java.util.Scanner* class has simplified console I0. Here is an example:

```
// File   : introductory/IntroScanner.java
// Purpose: Write to and read from the console.
// Author : Michael Maus
// Date   : 2005-03-29


import java.util.*;


public class IntroScanner

{

    public static void main(String[] args) {

        //... Initialization
        String name;                // Declare a variable to hold the name.
        Scanner in = new Scanner(System.in);

        //... Prompt and read input.
        System.out.println("What's your name, Earthling?");
        name = in.nextLine();       // Read one line from the console.

        //... Display output
        System.out.println("Take me to your leader, " + name);
    }
}
```

Here is a short summary of console IO.

Characteristic	**Console**
Imports	`import java.util.*; // Scanner`

Initialization	`// Declare and init Scanner object.`
	`Scanner input = new Scanner(System.in);`

Line of text input	`System.out.print("Enter your name: ");`
	`String name;`
	`name = input.nextLine();`

Integer input	`System.out.print("Enter your age: ");`
	`int age = input.nextInt();`

Output	`System.out.println(result);`

For console output, no imports are required. The `System` class is automatically imported (as are all `java.lang` classes). You can write one complete output line to the console by calling the System.out.println() method. The argument to this method will be printed. `println` comes from Pascal and is short for "print line". There is also a similar `print` method which writes output to the console, but doesn't start a new line after the output.

6. Lab Tasks

6.1. Study the corresponding Java JDK documentation for BigInteger and wrapper classes. Hint: Java documentation is installed on each lab computer Desktop in folder **j2se7docs.**

6.2. The legend says that the inventor of the game of chess asked a number of wheat grains: a double amount for each of the 64 squares of the board (i.e.1 for the first square, 2 for the second, 4 for the third, 2^{i-1} for the ith. Hint: use BigInteger class.

6.3. Write small programs to test the limitations of the representations. Note what happens if one:

- Adds an integral amount to the highest primitive value of each primitive type.
- Subtracts an integral amount to the highest primitive value of each primitive type.

- Uses floating point types to hold numbers with a number of decimal digits larger than the number of decimal digits which can accurately be represented in each of the two categories.
- Adds/subtracts amounts with a number of decimal digits larger than the maximum number of digits that can accurately be represented from a floating point number.

To accomplish this look in Java JDK documentation for wrapper classes (i.e. Byte, Short, Integer, Long, BigInteger, BigDecimal) static fields.