Introduction to Computer Programming

Using the Java Programming Language

2. Beginning Programming

This course utilizes the Java programming language to illustrate programming concepts. Please do not think that you are learning Java, when in reality you are learning how to program. Once you have mastered the concepts of programming, learning a new language is relatively straightforward. If you desire to learn Python, it will be a relatively simple task of moving to that language once this course is complete.

Focus on understanding each of the concepts that are presented in this course. Do not merely cut and paste examples and not understand what each line is doing. Using example programs to build upon is not a bad way of starting out, just be sure that you are not kidding yourself about understanding them. When a concept is used but not going to be taught until later in the course, the text will clearly state that the line must be used and it will be explained later. For all other lines in the program, you should understand their use before moving on to the next section.

Remember that programming builds upon previous concepts. So, if you don’t fully understand one section and decide to move on to the next section, you will not do well. We will use all of the concepts learned in a building block strategy so that our programs can become more and more complex as the course moves on. For this reason, truly master each section before moving on to the next. The best way to master a section is to write small programs that utilize the skills presented in that section.

Writing a Simple Program

There comes a time when every student must write their "hello world" program. Ask any programmer when they wrote their first hello world program and they will smile. They may not remember exactly when it was, but they surely did it! So, let's see what a "hello world" program looks like in Java.

```
1     /*
2      * Programmer: Mr. Kulla
3      * Date: January 1, 2020
4      * Purpose: This program will display Hello World to the console.
5      */

6     public class HelloWorld {

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

8                 System.out.println("Hello World");
9           }

10    }
```

Now let's see what all of these lines are for!

Lines 1-5: These lines are comments and are ignored by the compiler. Comments are placed in the program to make it easier for other humans to read it. They are not required to make the program run. The comment begins with line 1 and ends with line 5. Anything can be placed between lines 1 and line 5 and the compiler will ignore them. It is convention to place the *'s at the beginning of lines 2-4 because it looks nice, but they are not required. However, put them there, as this is the industry standard for a comment block like this. This comment block is required for every program you type in this course.

Line 6: Every Java program is a class and you must state the name of the class at the beginning of every Java program. The program file stored on your computer must be the same name as the class (including capitalization). This file on your computer would be saved as HelloWorld.java.

Line 7: A main method is needed so that the computer knows where to start running your program. In the future your programs will have many methods, so without a main method the computer would not know where to start. The concept of static will be discussed later. The main method takes one parameter that is required. The parameter is an array of Strings that we will discuss later. At this time of your programming, every one of your programs needs a main method declared in this exact way.

Line 8: The System.out.println() method is the basic output statement in Java and will display whatever is between quotation marks to the console.

How is Java Compiled?

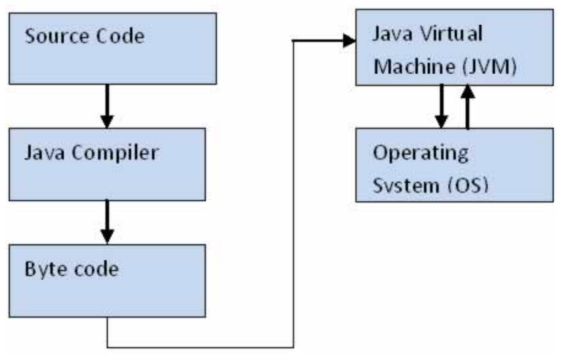
When we run our programs, the Java code gets compiled so that it can be run on your computer. Computers do not understand Java or any other high level language. These high level languages get compiled into a language that the computers do understand.

Java is slightly unique in that it does not get compiled into "machine language" that is understood on a computer. Each computer understands different machine languages because they use different processing chips. An Apple computer and an IBM computer speak two different machine languages.

Java got around this by creating a virtual machine called the Java Virtual Machine. This virtual machine is actually a program that is running on every computer. So, there's a JVM for a Mac, there's a JVM for an IBM, etc. When we compile our Java programs, they are understood by all JVM's, regardless of which type of machine they are running on.

Your source code (the program you type) is compiled using a Java compiler. This creates Byte Code. Byte Code is the language that JVM's understand. The JVM then converts that code into machine code for the machine that it is running on.

A diagram of this is shown here.



Keep in mind that although we typically work on our programs on the computer, sometimes it is often convenient to print them out. When someone requests a "hard copy" of your program, it means they want it printed out. If they request a "soft copy" of your program, it means they want an electronic copy.

Parts of a Program

Identifiers

Identifiers are the names of variables, methods, classes, packages and interfaces. Unlike literals they are not the things themselves, just ways of referring to them.

In the HelloWorld program, `HelloWorld`, `String`, `args`, `main` and `println` are identifiers.

Identifiers must be composed of letters, numbers, the underscore _ and the dollar sign `$`. Identifiers may only begin with a letter, the underscore or a dollar sign.

For this course, it is recommended that method names and class names capitalize the first letter of each word. For instance, these would be good method names:

```
CalculateSum()
PrintReport()
GetUserInput()
```

Variables

Java recognizes different data types of variables depending upon what kind of data they can contain. Java has eight built-in primitive data types designated by reserved words: **byte, short, int, long, float, double, char,** and **boolean.**

Variables of different types occupy different amounts of memory space and are described as having different sizes.

Of the eight primitive data types in Java, the four most commonly used for this introductory course are: **double, int, boolean,** and **char**.

- A variable is a named memory location that temporarily stores data that can change while the program is running.
- A final is a named memory location that temporarily stores data that remains the same throughout the execution of the program. It is a constant variable in the program. To make a variable a constant, insert the final keyword before it in the declaration:
```
    final int daysInWeek = 7;
```
- The type of a variable indicates what kind of value it will store.
- The name of a variable is known as its identifier.
- A variable is given a value through an assignment statement.

The following table illustrates the types of variables that are used in Java programs along with the values that they can hold.

Data Type	Java Keyword	Kind of Value	Bytes of Memory	Range of Values
Character	char	1 character	2	not applicable
Byte	byte	Integer	1	-128 to127
Short integer	short	Integer	2	-32,768 to 32,767 (-2^{15} to $2^{15} - 1$)
Integer	int	Integer	4	-2,147,483,648 to 2,147,483,647 (-2^{31} to $2^{31} – 1$)
Long Integer	long	Integer	8	-9223372036854775808 to 9223372036854775807 (-2^{63} to $2^{63} - 1$)
Float	float	Decimal values to 7 decimal digit precision	4	3.4e-38 to 3.4e38 positive and negative
Double	double	Decimal values to 15 decimal digit precision	8	1.7e-308 to 1.73e308 positive and negative
Boolean	boolean	Boolean (Logical) values True or False	1	True or false (0 or 1)

Naming Variables

Remember that the names of variables in the Java language are referred to as identifiers.

It is important to name your variables with meaningful names so that the reader of your program knows what values you are storing in it. When we name variables based on the content, they are known as mnemonic variables.

Let's take some examples. If I need to store the amount of money in my checking account, a good name for the variable may be `accountBalance`. This indicates to the reader that the balance of the account is stored in the variable.

It is unwise to name variables a, b, c, d, e, f, etc. in most cases, as these have no meaning to the reader of the program.

There are certain rules that must be adhered to when naming variables. They are:

- All variable names must begin with a letter of the alphabet, an underscore(_), or a dollar sign ($). In this course we will always use a letter of the alphabet as the first character.
- After the first initial letter, variable names may also contain letters and the digits 0 to 9. No spaces or special characters are permitted.
- Variable names can be any length, but try to be concise. Remember that you will need to type the variable name in your code every time you use it.
- Uppercase characters are distinct from lowercase characters.
- Variable names are case-sensitive.
- Use ALL capital letters for constants (final).
- You cannot use a java reserved word for a variable name.

For our programs in this course, we will be using capital-style notation for all variable names and each variable will start with a lower case letter. Examples of valid variable names would be:

```
accountBalance
gradePointAverage
numberOfBooks
cellPhoneCost
```

Declaring Variables

Remember that all variables must be declared before they can be used in your program. For this course, under most circumstances, all variables will be declared at the top of your method or class. This makes finding the variable declarations easier for the reader of the program.

To declare a variable, follow these simple steps:

- Choose the type of variable you need (integer, decimal, character, string, etc).
- Choose a good name for the variable following the course naming standards.
- Use the following format for a declaration statement:

  datatype variable identifier;
- You can declare more than one variable on a line if they are the same data type. To do this, we separate the variable names with commas as seen here:
  ```
  int age, weight, height;
  ```
- You may initialize a variable (give it a default value) in a declaration statement. Do not assume that your variables will have initial values assigned to them. For instance, if you declare a variable `counter`, do not assume it starts at zero. These uninitialized variables can contain garbage that was leftover from another program that used the memory before you declared your variable. You must initialize variables if you plan to use the value before storing a value into it.
  ```
  double amount=2.75;
  ```

When you declare a variable, a space in the computer memory is being allocated to hold the values for that value. These spaces that are reserved to hold the value are different sizes based on the data type assigned to the variable.

Declaring Constants

When you will be using constants in your program, you should declare them as final's. This allows the reader of the program to quickly see that the variable is a constant and it's value will not change. The following format is used to declare a constant:

```
     final int DOZEN=12;
```

Reserved Words

The following is a list of reserved words for Java. You may not redefine any of these reserved words. Therefore you cannot name your methods, classes, variables or any other identifier the same as any of these words. Note that these words will always appear in lowercase.

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

In addition, there are 3 reserved words for literal values.

true	null	false

Assignment Statements

To store values into variables we use the assignment statement. The assignment statement uses an equal sign (=) to place a value into a variable. The format for an assignment statement is:

variable = *expression*;

The *variable* is any name that of a variable that you have previously declared in your program. The *expression* is a variable, numerical/literal, or an expression that produces the data type that is the same as the variable you are assigning into. Keep in mind that you can store data in a variable at any time after it is declared. We previously assigned data to a variable immediately after it was declared as in:

```
int length = 15;
```

This could have been accomplished in two separate lines of code. We could have declared the variable at the top of the program, and then initialized it at some point further down in our program:

```
int length;
<more program lines>
length = 15;
```

Assignment statements do not always need to be storing a literal or value into a variable. They could be performing calculations. For instance, to calculate the area of a rectangle in Java, the code would look something like:

```
    int length = 10;
    int width = 25;
    int area;

    area = width * length;
```

The built-in mathematical operators that can be used in an expression are:

Operator	**Description**
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

The majority of these should be clear to you with the exception of modulus. Modulus is the remainder dividing the first value by the second. So:

`5%2` = 1 (2 goes into 5 twice, with a remainder of 1)
`10%4` = 2 (4 goes into 10 twice with a remainder of 2)
`23%5` =2 (5 goes into 23 four times with a remainder of 3)

Division is also a bit tricky in Java, compared to mathematics, because it depends on the type of the variables or numbers being used. When we are performing integer division (dividing two integers), the result is an integer, truncated, not rounded. Look at the following cases for division:

Type of Division	**Example**	**Result**
Integer	`4/3`	1
Integer	`10/4`	2
Mixed	`11.0/5`	2.2
Double	`11.0/5.0`	2.2

Order of Operations applies in Java the same way it applies in math. So, an expression such as: `2+4*3` would result in 14, not 24.

Augmented Assignment Statements

In computer programming it is very useful to do a calculation that uses a value of a variable and store it back into that variable. For example, we may want to increment the length of a side of a rectangle by 10. To do this, the statement would look like:

```
      length = length + 10;
```

This statement takes the value of length, adds 10 to it, and then stores it back into length.

It is also useful to increment or decrement a variable by 1. We do this when we are counting, and in computer programs we count quite often. To increment a counter by 1, we would use the following statement:

```
      counter = counter + 1;
```

Fortunately Java provides us with a compound assignment operator to shorten these types of statements. The following compound assignment operators are provided:

Operator	**Description**
+=	Addition and assignment
-=	Subtraction and assignment
*=	Multiplication and assignment
/=	Division and assignment
%=	Modulus and assignment

Using these compound assignment operators, we can change the line of code:

```
      counter = counter + 1;
```

to the line of code:

```
      counter += 1;
```

Additional example:

```
      int x = 3;
      x+=5;
```

This would yield x to be 8 and is equivalent to the following lines of code:

```
      int x = 3;
      x = x + 5;
```

Both methods are valid in Java. Use whichever methods you feel comfortable with. This is merely a shorthand notation for something that we do quite often when coding.

Comments

Comments are mandatory for good code. Do not assume that another reader of your code, or even you, the original programmer, will know what the code is doing months or years from now. Comments are essential so that your code can be maintained in the future.

There are 3 styles of comments that are used within a program:

`/** documentation */`	documentation commenting.
`/* text */`	block commenting.
`// text`	single line commenting.

Comment statements are ignored by the compiler. Their only purpose is for the human reader of the code.

Examples:

1. Header for all programs:
All programs should begin with a comment block that states the purpose of the program, the programmer, and the date it was created. Every exercise in this course requires a header in the following format.

```
/*
 * Programmer: Mr. Kulla
 * Date: January 1, 2020
 * Purpose: This program will display Hello World to the console.
 */
```

2. Inline comments:
Programs should include comments within the program. Mostly these comments are short and are there to help the reader understand what a block of code is doing. They can be place on the same line as the code, or above it.

```
public static void main(String[] args) {
     System.out.println("Hello World"); // Print a line to the console
}

public static void main(String[] args) {
     // Print a line to the console
     System.out.println("Hello World");
}
```

3. Longer comments within the code:
Remember that you can use the header style comments within the code for longer comments.

```
public static void main(String[] args) {
     /* Print a line to the console
        This is done by using the println method.
     */
     System.out.println("Hello World");
}
```

Or you could continue to use the inline format to achieve the same results:

```
public static void main(String[] args) {
     // Print a line to the console
     // This is done by using the println method.
     System.out.println("Hello World");
}
```

Reading and Writing to the Console

The majority of the initial programming we do in this course will read and write from the console. No windows programming will be done until we are half way through the course and fully understand more of the intricate concepts of programming in an object oriented language such as Java. Therefore it is important that you master how to read and write to the console.

Writing to the Console

We have already seen how to write to the console using the `println` method. In addition to the `println` method, we have the `print` method. The only difference between the two methods is that the `println` method places a carriage return at the end of it's text that is displayed. Therefore, if you use two `println` calls one after the other, the lines of text will be displayed on separate lines. The use of the `print` method does not place a carriage return at the end, and therefore two `print` methods called one after the other would result on the text being displayed on the same line. This can be seen in the following example:

```
/*
 * Programmer: Mr. Kulla
 * Date: January 1, 2020
 * Purpose: This program will display a few lines to the console.
 */

public class HelloWorld {
      public static void main(String[] args) {
            System.out.println("This is the first line.");
            System.out.print("This is the second line.");
            System.out.println(" This is also the second line.");
      }
}
```

Output from the above program would look like this:

```
This is the first line.
This is the second line. This is also the second line.
```

Notice how the use of the `print` method did not move to the next line after it displayed it's text.

Reading from the Console

There is only so much your program can do if you are not going to ask the user any questions in your program. To do this we need a way for the program to stop and wait for the user to enter in an answer to a question. One of the easiest ways of accomplishing this is by using the Scanner class. There are many ways to read from the console, but for what we need to do in this course, the Scanner class will provide us with a very simple method.

The methods that we will use to read in values from the console are:

Method	**Description**
next()	Returns the next token from the scanner as a String.
nextLine()	Returns an entire line as a String.
nextShort()	Returns the next token as a short value.
nextInt()	Returns the next token as an int value.
nextLong()	Returns the next token as a long value.
nextFloat()	Returns the next token as a float value.
nextDouble()	Returns the next token as a double value.

A sample program showing how to read from the console is given here. This example reads an integer value from the console.

```
      /*
       * Programmer: Mr. Kulla
       * Date: January 1, 2020
       * Purpose: This program will show how to read from the console.
       */

1     import java.util.Scanner;

      public class ConsoleInput {

            public static void main(String[] args) {

2             Scanner in = new Scanner(System.in);
3             int num;

              System.out.println("Please enter a number:");
4             num = in.nextInt();

5             System.out.println("You entered: " + num);

6             in.close();
            }
      }
```

Let’s look at what some of the new lines in this program do.

Line 1: Imports the scanner class so that we can use all of the methods in that class in our program. We are only going to use the `nextInt` method for this example program, but need to import the class to use it.

Line 2: Declares a variable `in` that we will use to read from the console. When we want to read from the console we use `System.in` as a parameter to the Scanner method. Later when we read from files, we will change this. At this point in your programming, use this line as is. Feel free to change the variable name from `in` if you prefer a different name.

Line 3: Declares an integer variable `num`.

Line 4: Pauses the programming waiting for the user to type in an integer value. Once the user types in the integer value and hit's Return, the value will be stored in the variable `num`.

Line 5: Displays the number that the user typed in to the console.

Line 6: Closes the input stream. Although this is not mandatory when reading from the console, it is good practice to always perform a close whenever you open a stream as we did in line 2 of this program.

The output from this program would look like:

```
Please enter a number:
3
You entered: 3
```

Escape Sequences for Println

Often it is useful to use Escape sequences to format when you are writing to the console. The following Escape sequences can be used in your program, either in the `print` or `println` methods. These Escape sequeces are placed between the quotation marks in your method call.

Escape Sequence	Name	Description
`\n`	New line	Moves to the beginning of the next line
`\t`	Horizontal tab	Moves to next tab position. Note that tab spacing is every 8 columns starting with 1. (Columns 9, 17, 25, 33, 41...)
`\\`	Backslash	Displays a backslash
`\'`	Single quote	Displays a single quote
`\"`	Double quote	Displays a double quote

Example 1:

```
System.out.println("Hello \nGoodbye \nAll Done.");
```

Displays:

```
Hello
Goodbye
All Done.
```

Example 2:

```
System.out.println("A\tB\tC");
```

Displays:

```
A       B       C
```

Println vs. Printf

`Println` is very convenient, especially when we are displaying Strings. However, we often need to format strings and numbers so that they fit into nice columns. We also may want to format numbers so they display the way we would like. Money is a common thing to display, yet we must format a decimal number to have it display as money ($12.50 instead of $12.5). For this type of formatting, it is easiest to use the `printf` method.

`Printf` utilizes % signs to denote a format specifier. The format specifecers that we will use most in this course are as follows:

Specifier	Description
%c	character
%d	decimal (integer) number (base 10)
%e	exponential floating-point number
%f	floating-point number
%i	integer (base 10)
%o	octal number (base 8)
%s	a string of characters
%u	unsigned decimal (integer) number
%x	number in hexadecimal (base 16)
%%	print a percent sign

Let's take a look at two examples. First, we will format a small table so that all columns line up.

```
public class PrintFExample {

      public static void main (String[] args) {
            int x = 5;
            int y = 100;
            String s1 = "Hello";
            String s2 = "Goodbye";

            System.out.printf("%15s %12s %n", "Column 1", "Column2");
            System.out.printf("%15d %12s %n", x, s1);
            System.out.printf("%15d %12s %n", y, s2);
      }
}
```

This program displays three lines to the console. It places data for two columns, the first column 15 characters wide, and the second column 12 characters wide.

Output from this program would be:

```
       Column 1      Column2
              5        Hello
            100      Goodbye
```

The next example will format money, which is something we often need to do. Remember that a floating point number 12.5 is not in the correct format for money. We need the trailing zero so that it will appear as 12.50. The easiest way to do this is by using the `printf` method.

```
public class PrintFExample2 {

      public static void main (String[] args) {
            double amount = 12.5;

            System.out.println("Displayed Wrong: $" + amount);
            System.out.printf("Displayed Correct: $%.2f", amount);
      }
}
```

Output from this program appears here:

```
Displayed Wrong: $12.5
Displayed Correct: $12.50
```

There are many uses of `printf` formatting. Here are some examples for you to use in your code.

Controlling integer width with `printf`
The %3d specifier means a minimum width of three spaces, which, by default, will be right-justified:

`printf("%3d", 0);`	`0`
`printf("%3d", 123456789);`	`123456789`
`printf("%3d", -10);`	`-10`
`printf("%3d", -123456789);`	`-123456789`

Left-justifying `printf` integer output
To left-justify integer output with printf, just add a minus sign (-) after the % symbol, like this:

`printf("%-3d", 0);`	`0`
`printf("%-3d", 123456789);`	`123456789`
`printf("%-3d", -10);`	`-10`
`printf("%-3d", -123456789);`	`-123456789`

The `printf` integer zero-fill option
To zero-fill your printf integer output, just add a zero (0) after the %symbol, like this:

`printf("%03d", 0);`	`000`
`printf("%03d", 1);`	`001`
`printf("%03d", 123456789);`	`123456789`
`printf("%03d", -10);`	`-10`
`printf("%03d", -123456789);`	`-123456789`

`printf` Integer Formatting

As a summary of printf integer formatting, here's a little collection of integer formatting examples. Several different options are shown, including a minimum width specification, left-justified, zero-filled, and also a plus sign for positive numbers.

Description	Code	Result
At least five wide	`printf("'%5d'", 10);`	`'   10'`
At least five-wide, left-justified	`printf("'%-5d'", 10);`	`'10   '`
At least five-wide, zero-filled	`printf("'%05d'", 10);`	`'00010'`
At least five-wide, with a plus sign	`printf("'%+5d'", 10);`	`'  +10'`
Five-wide, plus sign, left-justified	`printf("'%-+5d'", 10);`	`'+10  '`

Formatting Floating Point Numbers with `printf`

Here are several examples showing how to format floating-point numbers with printf:

Description	Code	Result
Print one position after the decimal	`printf("'%.1f'", 10.3456);`	`'10.3'`
Two positions after the decimal	`printf("'%.2f'", 10.3456);`	`'10.35'`
Eight-wide, two positions after the decimal	`printf("'%8.2f'", 10.3456);`	`'   10.35'`
Eight-wide, four positions after the decimal	`printf("'%8.4f'", 10.3456);`	`' 10.3456'`
Eight-wide, two positions after the decimal, zero-filled	`printf("'%08.2f'", 10.3456);`	`'00010.35'`
Eight-wide, two positions after the decimal, left-justified	`printf("'%-8.2f'", 10.3456);`	`'10.35   '`
Printing a much larger number with that same format	`printf("'%-8.2f'", 101234567.3456);`	`'101234567.35'`

`printf` String Formatting

Here are several examples that show how to format string output with printf:

Description	Code	Result
A simple string	`printf("'%s'", "Hello");`	`'Hello'`
A string with a minimum length	`printf("'%10s'", "Hello");`	`'     Hello'`
Minimum length, left-justified	`printf("'%-10s'", "Hello");`	`'Hello     '`

Style & Format

Computer programs should be written so that they can be read, understood, and modified by human readers. Programming style is important, and most programs are evaluated for style as well as for correctness. This page lists some basic programming style rules that you should keep in mind as you write your programs. The rules for methods and for classes will only be applicable after those topics have been covered in this course.

The Main Rule

- A program should be readable. All the other rules on this page can be overridden by this primary rule. Your programs should display good taste, which you can generally acquire only by paying attention to the practices of other, expert programmers and the examples given in this course.

Comments

- Every class should have a comment block header that specifies the purpose of the class. This should include the date it was written, the programmer name, and a brief explanation.
- Every variable that has a non-trivial role in the program should have a comment that explains its purpose.
- Comments can be included in the body of a method when they are needed to explain the logic of the code.
- Comments should never be used to explain the Java language. A comment such as "declare an int variable named x" or "increment the variable ct" is worse than useless. Assume that your reader knows Java! (Note that such comments are sometimes used in programming textbooks or on the blackboard, but never in real programs.)

Formatting

- Use indentation to display the structure of your program. The body of a class definition should be indented. The body of a method definition should be indented. When a statement is nested inside another statement, it should be indented one additional level. (Note that Eclipse can fix the indentation of any segment of code: Just hilite the code segment and hit Command-I.)
- An ending "}" should be on a line by itself. The opening "{" can be at the end of a line, or it can be placed on a line by itself so that it lines up with the matching "}".
- Don't put more that one statement on a line.
- Avoid very long lines. In general, lines should not be longer than 80 characters. Longer statements should be broken up over several lines.
- Avoid very deep nesting of statements. If you find yourself using more than two or three levels of nesting, think about defining some subroutines in order to break up your code into more manageable chunks.
- Blank spaces and blank lines can make a program easier to read. In general, you should put some blank lines between method definitions. Leave spaces around operators such as =, ==, !=, and so on.

Naming

- Use meaningful names for your variables, methods, and classes.
- Follow the usual Java convention for capitalization: Variable names begin with lower case letters. Class names begin with upper case letters. Method names can begin with either, but I prefer upper case letters. When a name contains more than one word, capitalize the extra words, as in "interestRate".
- Use final to declare named constants to represent constant data. Constants generally have names that are entirely in upper case, with words separated by underscore characters.

Methods

- A method should have a clear, single, identifiable task.
- An individual method definition should not be too long. In general, a function should not be longer than one printed page (and that is stretching it).
- Instance methods can access the instance variables that represent the state of an object, but you should avoid using instance variables simply to pass information from one method to another -- for that, you should use parameters and return values.

Classes

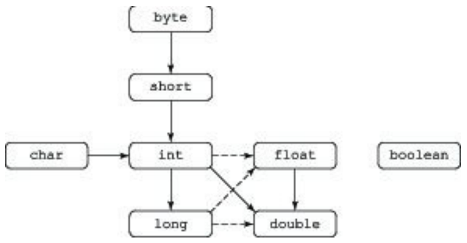
- A class should represent a clear, single, identifiable concept.
- Use the modifiers public, protected, and private to control access to the variables and methods in a class.
- Member variables should, in general, be declared to be private. Getter and setter methods can be provided to access and manipulate the private member variables.

Numeric Type Conversion

From time to time, you need to convert numeric data of one type to another in Java. You may need to convert a double value to an integer, or vice versa. Some conversions can be done automatically; others are done using a technique called casting.

Automatic Type Conversion in Java

Java can automatically convert some primitive types to others and do so whenever necessary. The image below shows which conversions Java allows. Note that the conversions shown with dotted arrows below may cause some of the value's precision to be lost. An `int` can be converted to a `float`, for example, but large `int` values won't be converted exactly because `int` values can have more digits than can be represented by the `float` type.



Whenever you perform a mathematical operation on two values that aren't of the same type, Java automatically converts one of them to the type of the other. Here are the rules Java follows when doing this conversion:

- If one of the values is a `double`, the other value is converted to a `double`.
- If neither is a `double` but one is a `float`, the other is converted to a `float`.
- If neither is a `double` nor a `float` but one is a `long`, the other is converted to a `long`.
- If all else fails, both values are converted to `int`.

Type Casting

Casting is similar to conversion but isn't done automatically. If you want to convert a `double` to an `int`, for example, you must use casting.

When you use casting, you run the risk of losing information. A `double` can hold larger numbers than an `int`, for example. In addition, an `int` can't hold the fractional part of a `double`. As a result, if you cast a `double` to an `int`, you run the risk of losing data or accuracy, so 3.1415 becomes 3, for example.

To cast a primitive value from one type to another, you use a cast operator, which is simply the name of a primitive type in parentheses placed before the value you want to cast. For example:

```
double pi = 3.1314;
int iPi;

iPi = (int)pi;
```

Note that the fractional part of a `double` is simply discarded when cast to an `int`; it isn't rounded. For example:

```
double price = 9.99;

int iPrice = (int)price;
```

Here `iPrice` is assigned the value 9. If you want to round the `double` value when you convert it, use the round method of the Math class.

String Type

The `String` Class is provided by Java and allows for the programmer to easily store more than one character at a time. In Java, strings are objects, not primitive types such as `int`, `double`, `char`, and `boolean`. Many people believe that the `String` type is a native Java type, but in reality it is not. Thus the reason that the `String` type uses a capital S whereas all other native types in Java start with lower case characters.

The term string literal refers to any alphanumeric (character, digit, or both) enclosed in double quotation marks (such as "Hello World" or "I love to program in Java").

Be very careful about storing numbers in a String type. You can do this, but it is no longer a number! If a string literal contains only numeric digits ("12345") it is not a number, it is just a string of numeric digits that cannot be used to perform mathematical calculations such as addition or multiplication. Remember that you can perform calculations on numeric data only, not on string literals.

Declaring a String variable is different from declaring an int, double, char or boolean. The variable name of a String holds the address of the memory location where the actual body of the string is stored. Variables that are used in this manner are called references. The address values stored in these variables are said to point to the actual data and not contain the actual data themselves.

Let's take a look at a simple segment of code to use a `String` type.

```
      String stuffToPrint = "Hello World";
      System.out.println(stuffToPrint);
```

The code above would result in the following output to the console:

```
      Hello World
```

It is possible to concatenate strings in your program with the + operator. This can be seen in the following code segment:

```
      String word1 = "Hello";
      String word2 = "World";
      System.out.println(word1 + " " + word2);
```

The code above would result in the following output to the console:

```
      Hello World
```

Since `String` is a class and not just a data type, it also includes methods that we can use once we declare a variable of type `String`. Some of the more common ones that we will use in this class, along with examples, are provided here.

toUpperCase() and toLowerCase()

Code	**Results**

`String s="Sachin";`
`System.out.println(s.toUpperCase());`
	`SACHIN`

`String s="Sachin";`
`System.out.println(s.toLowerCase());`
	`Sachin`

`String s="Sachin";`
`System.out.println(s);`
	`Sachin`

trim()

Code	**Results**

`String s="  Sachin  ";`
`System.out.println(s);`
	`  Sachin`

`String s="  Sachin  ";`
`System.out.println(s.trim());`
	`Sachin`

startsWith() and endsWith()

Code	**Results**

`String s="Sachin";`
` System.out.println(s.startsWith("Sa"));`
	`true`

`String s="Sachin";`
` System.out.println(s.endsWith("n"));`
	`true`

charAt()

Code	**Results**

`String s="Sachin";`
`System.out.println(s.charAt(0));`
	`S`

`String s="Sachin";`
`System.out.println(s.charAt(3));`
	`h`

* Remember that strings start numbering at 0, not 1!

length()

Code	**Results**

`String s="Sachin";`
`System.out.println(s.length());`
	`6`

valueOf()

Code	**Results**

`int a=10;`
`String s=String.valueOf(a);`
`System.out.println(s+s+s);`
	`101010`

indexOf()

Code	**Results**

`String address = "Cupertino, California 95104";`
`System.out.println (city.indexOf(","));`
	9

equals()

Code	**Results**

`String userName = "Bob";`
`System.out.println(userName.equals(“Bob”));`
	true

equalsIgnoreCase()

Code	**Results**

`String userName = "Bob";`
`System.out.println(userName.equalsIgnoreCase(“bob”));`
	true

Sample Program

This sample Java program and the notes are used to demonstrate some of the ideas taught in this unit. The line numbers are not part of the Java program - they are used for reference purposes only.

```
1    /*
      * Programmer: Mr. Kulla
      * Date: January 1, 2020
      * Purpose: Demonstrate some concepts from this unit.
      */

2    import java.util.Scanner;

3    public class EbayPurchase
4    {

5          public static void main(String[] args)
6          {
7                Scanner input = new Scanner(System.in);

8                final double FLAT_SHIPPING = 7.95;
9                double itemCost, totalCost;
10               int number;

11               System.out.println("Enter the cost of one item");
12               itemCost=input.nextDouble();
13               System.out.println("Enter the number of items purchased");
14               number = input.nextInt();
15               totalCost=number*itemCost + FLAT_SHIPPING;
16               System.out.println("The total bill will be "+ totalCost);

17               input.close();
18         }
19   }
```

Line 1: Leading documentation statements. Remember that comment statements are ignored by the compiler. Comments are intended to be messages to the reader of the code. While a program should be somewhat "self-documenting", well placed comments enhance the reader's understanding of the code (and oftentimes your own understanding).

Line 2: `import java.util.Scanner` is included to deal with input (from the console in our case).

Line 3: Remember that the name of the class must be the same as the name of the file containing your Java code. In this situation, the program code was saved as `EbayPurchase.java`. Every Java program is a class.

Line 4: Brackets { } are needed for all classes.

Line 5: The main method indicates where execution will start. Remember that Java is case sensitive and this line must be typed correctly.

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

Line 6: Brackets { } are needed for all methods.

Line 7: Declare a new variable `input` that will be used to read from the console. `System.in` informs the variable that the input will be typed by the user (vs. coming from a file).

```
Scanner input = new Scanner(System.in);
```

Line 8: The reserved word `final` is used in Java to declare a constant variable.

```
final double FLAT_SHIPPING = 7.95;
```

A `double` is used here to allow for a decimal value. Remember that variable names for constants are upper case.

Line 9: These variables are declared as `double` since they will contain decimal values.

```
double itemCost, totalCost;
```

Line 10: The number of items purchased is declared as an `int` value.

```
int number;
```

Lines 11-14: We are using the input routines from the `Scanner` class to receive a `double` value entry and an `int` value entry from the user. We are also using the `println` method to prompt the user.

Each of these commands will prompt the user as to what is needed and then store the answers in the variables `itemCost` and `number`, respectively.

Line 15: The mathematical computations are done in this line. The cost for the stated number of items is computed and the shipping fee is added. The final answer is stored in the variable `totalCost`.

```
totalCost=number*itemCost + FLAT_SHIPPING;
```

Lines 16: Printing to the screen. This statement will print the message "The total bill will be" followed by the price that was computed in the previous line.

```
      System.out.println("The total bill will be "+
totalCost);
```

Line 17: Whenever we are reading input from the console or a file, we must *close* the stream of input when we are done.

```
input.close();
```

Lines 18 and 19: These lines close the brackets for the method and class.