Chapter 2

Elementary Programming
Variables and Expressions

Programs basically manipulate data values (numeric and/or text). These values must be stored in memory so the CPU can process them. In order to facilitate easy access to these memory locations, variables are used. Using variables allows a programmer to read, write and manipulate the values in memory with ease and efficiency.
A variable names a memory location

<table>
<thead>
<tr>
<th>Memory address</th>
<th>Variable Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>yearsMarried</td>
</tr>
<tr>
<td>996</td>
<td>numberOfChildren</td>
</tr>
</tbody>
</table>

Recall, each memory location has a unique address, which is of course numeric. Since numbers are good for computers, but not for humans, variable names are used instead.

In effect, a variable name simply represents some memory location. The advantage is that a programmer can now use the variable's name to access the value in memory.
A variable's value can change

A variable is something that can change, and indeed the contents of each variable will undoubtedly change through the course of execution.
In the program above, four variables are declared, each one of type int. A variable of type int can hold whole numbers, both positive and negative.

The variables are declared and initialized, giving each one an initial value.

Variables should be named in such a way that their name reflects the purpose of the variable or the type of data it stores. Avoid variable names that are single characters or that are vague.
System.out.println("Your total amount is $" +
    totalDollarAmount);
System.out.println("$50 x " + numberOfFiftyDollarBills +
    " = "$ +
    numberOfFiftyDollarBills * 50);
System.out.println("$20 x " + numberOfTwentyDollarBills
    " = "$ +
    numberOfTwentyDollarBills * 20);
System.out.println(" $1 x " + numberOfOneDollarBills +
    " = "$ +
    numberOfOneDollarBills);

Your total amount is $285
$50 x 4 = $200
$20 x 3 = $60
$1 x 25 = $25
A data type determines how much memory is allocated.

A variable declaration allows the compiler to decide how much memory the variable should occupy in memory.

Different data types require different amounts of memory, and the number of bytes allocated for the variable define the maximum value that can be stored within that memory location.
Primitive types are used for variables

Java uses int and long for whole numbers, float and double for real numbers, char for single characters and boolean for true/false values.

Each data type requires its own unique translation to binary digits (0 and 1) before stored in memory. Recall that a computer can only process strings of 0s and 1s, so everything eventually must get translated to binary.

A primitive variable holds its actual value.
Variable of a class type, referred to as objects, simply hold a reference to the actual object itself. Therefore, objects refer to their values indirectly.
An identifier is anything you name

```java
int numberOfEggs;  // Variable name

public class EggBasket {...}  // Class name

public static void main (...) {...}  // Method name
```
Identifiers may begin with a letter or underscore

_numEggs Variable: Capitalize subsequent words

class EggBasket

EggBasket Class: Capitalize each word

class main

main Method: Capitalize subsequent words

The rules for capitalizing are conventions, not compiler rules, so you could deviate, but that would not make most of the readers of your code happy.
Identifiers may contain letters, digits or underscore

Valid:      anotherDay
            _yesOrNo
            no_one_knows
            easyAs123

Invalid:    another Day
            ?yesOrNo
            no.one.knows
            123Abc
            if

Note the invalid variables:
- another Day: This variable contains a space
- ?yesOrNo: Starts with a question mark
- no.one.knows: Contains periods
- 123Abc: Starts with a digit
- if: This is a reserved word in the Java language.
Identifiers are case sensitive

All of the following variables are different.

```java
int myNumber;
int MyNumber;
int mynumber;
```

In Java identifiers are case sensitive, thus you must be a good speller. If you accidentally misspell one of your variables, Java will think you are referring to a totally different variable than the one you declared. This can cause compilation errors.
A variable can be assigned a value at any time. You should use the assignment operator (=) to assign a value to your variable.

Declared variables are not necessarily given initial values, so make sure to always assign them a value before using them. If you try using a variable that has not been assigned a value, thus it is uninitialized, the compiler will complain.

Note: A variable can be assigned a value at declaration; this is called initializing the variable.

```c
int sum;
int num1;
int num2;
char firstInitial;

num1 = 10;
num2 = 20;
sum = num1 + num2;
firstInitial = 'S';
```
import java.util.Scanner;  // Import the Scanner class

public class SampleIO {
    public static void main(String[] args) {
        // Declare Scanner object.
        Scanner keyboard = new Scanner(System.in);
        // Read the value from the keyboard.
        
        value = keyboard.nextInt();
    }
}

import java.util.Scanner; // Import the Scanner class

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

        // Read the value from the keyboard.
        System.out.println("Enter an integer value: ");
        ... 

        // Output the value just read in.
        System.out.println("You entered the number: " + value);
    }
}
import java.util.Scanner; // Import the Scanner class

public class SampleIO {
    public static void main(String[] args) {
        // Declare Scanner object.
        Scanner keyboard = new Scanner(System.in);
        // Read the value from the keyboard.
        System.out.println("Enter an integer value: ");
        int value = keyboard.nextInt();
        // Output the value just read in.
        System.out.println("You entered the number: " + value);
    }
}
Use constants for values that do not change

double pi = 3.14;

// Change the value of the variable pi.
pi = 3.14159;

public static final double PI = 3.14159;

// This will cause an error, since PI is not allowed to change.
PI = 3.14;

public static final int DAYS_PER_WEEK = 7;

Constants are declared and initialized inside the class, but outside any method. This makes them accessible to all methods inside the class.

The keyword final states that this is the final value this variable will receive, and that of course implies it can never change again. Cool.
Be aware of type incompatibilities

- The type of the **variable** should match the type of the **value** assigned to it.

- The compiler will make the necessary conversions.

- Smaller types will convert to larger types (widening).

Java like many other languages don't like to mix and match their data types. This of course has to do with the memory representation of each type. Remember, the CPU, which ultimately processes the numbers or text, can only perform operations on similar type values. So, one value must be converted to match the other before the CPU can process them.

There are two types of conversions. The first is the conversion of a smaller type to a larger type, called a widening conversion, and is automatically performed by the compiler. The second is from a larger type to a smaller type, and this requires a type cast.
The first conversion is performed automatically, since 7 (an int) is assigned to a larger type (double).

The second is invalid since the conversion is a narrowing one (double to an int).

The third is once again a widening conversion (int to double), and the type cast is only for emphasis and clarity, and highly recommended, not required.

The fourth is a required type cast. It is the only way to perform a narrowing conversion.
Can you say arithmetic?

+ addition
- subtraction
* multiplication
/ division (quotient)
% modulus (remainder)
Arithmetic is performed in pretty much the normal way, with a few exceptions.

First, remember that operations can only be performed on values of the same type, so a conversion to a larger type (widening conversion) may have to be performed first. Then, the evaluation takes place and then the result is assigned.

Notice how in the second example the result of the expression on the right of the assignment was 12.5, but since this value is assigned to an integer variable, the final value stored in the variable is 12.
Division is tricky

```java
int result = 10 / 5;
result: 2

int result = 1 / 2;
int result = 0;
result: 0

double result = 1 / 2.0;
double result = 1.0 / 2.0;
result: 0.5
```

Division is special in Java. Division of integer values always results in an integer result, regardless of whether it is assigned to an integer or a double variable (see the second example above).

If one or both of the arguments is of a floating point type (float, double) then the result is a real value (see third example).

One consequence of this behavior is demonstrated in the second example, where 1 / 2 leads to 0, not 0.5 as might have been expected. Be careful of this.
The modulus is the remainder basically of a division operation. If the / gives the quotient, then % gives the remainder.

The modulus is also special when dealing with integer values. In the second example, 1 % 2 produces 1, not 0, since 2 does not divide into 1. Just remember, the denominator (2) must divide into the numerator (1) as a whole number.
Follow the rules of precedence

unary: +, -, !, ++, -- right to left evaluation

binary: *, /, % left to right evaluation

binary: +, - left to right evaluation
Follow the arithmetic rules

```c
int result = 10 + 5 - 3 * 4;
int result = 10 + 5 - 12;
int result = 15 - 12;
int result = 3;
result: 3
```

The modulus is the remainder basically of a division operation. If the / gives the quotient, then % gives the remainder.

The modulus is also special when dealing with integer values. In the second example, 1 % 2 produces 1, not 0, since 2 does not divide into 1. Just remember, the denominator (2) must divide into the numerator (1) as a whole number.
The modulus is the remainder basically of a division operation. If the / gives the quotient, then % gives the remainder.

The modulus is also special when dealing with integer values. In the second example, 1 % 2 produces 1, not 0, since 2 does not divide into 1. Just remember, the denominator (2) must divide into the numerator (1) as a whole number.
Parentheses always win

```c
int result = 10 + (5 - 3) * 4;
int result = 10 + 2 * 4;
int result = 10 + 8;
int result = 18;
result: 18
```

Use parentheses to change the order of evaluation. All parenthesized expressions are evaluated first, starting with the most deeply nested.

In this example, the subtraction is now performed before the multiplication, since we used parentheses around the expression 5 - 3.
Feel free to specialize the assignment operator

```javascript
result = result + 10;
result += 10;

var -= value;  // var = var – value
var /= value;  // var = var / value
var *= value;  // var = var * value
var %= value;  // var = var % value
```

If you prefer to use a more concise form of the arithmetic operators, you may use the operators shown above. They are simpler to write and often used by programmers.

Note that the arithmetic operator comes before the assignment operator. Also keep in mind that the right hand side could be a full blown expression. In that case the expression will first be evaluated and then operated on.
Vending Machine Change
<Algorithm>

1) Read the amount in cents into the variable amountInCents.
2) Save amountInCents: originalAmountInCents = amountInCents;
3) Determine # of quarters: quarters = amountInCents / 25;
4) Reset amountInCents: amountInCents = amountInCents % 25;
5) Determine # of dimes: dimes = amountInCents / 10;
6) Reset amountInCents: amountInCents = amountInCents % 10;
7) Determine # of nickels: nickels = amountInCents / 5;
8) Reset amountInCents: amountInCents = amountInCents % 5;
9) Determine # of pennies: pennies = amountInCents % 5;
10) Display originalAmount and qty of each coin.

Use of an algorithm can be handy in problem solving. Writing your steps first before coding, makes the coding part so much easier. I encourage you to use algorithms for your coding problems, as it will hone your problem solving skills.
Application Deconstructed

<ChangeMaker.java>

```java
import java.util.Scanner;

public class ChangeMaker {
    public static final int CENTS_PER_QUARTER = 25;
    public static final int CENTS_PER_DIME = 10;
    public static final int CENTS_PER_NICKEL = 5;

    public static void main(String[] args) {

        int amountInCents;
        int originalAmountInCents;
        int quarters;
        int dimes;
        int nickels;
        int pennies;
```

Once the algorithm is developed it can then be translated to Java, which should be a straight forward step, since the logic is already expressed in the algorithm.

Most often the translation will be one to one, between steps in the algorithm and lines in the code.
Application Deconstructed
<ChangeMaker.java>

// Ask for the amount in cents.
System.out.print("Enter an amount of pennies [1-99]: ");

// 1) Read the amount in cents into the variable
amountInCents.
Scanner keyboard = new Scanner(System.in);

amountInCents = keyboard.nextInt();
// 2) Save the amount for later.
originalAmountInCents = amountInCents;

// 3) Determine # of quarters.
quarters = amountInCents / CENTS_PER_QUARTER;

// 4) Reset the amountInCents.
amountInCents %= CENTS_PER_QUARTER;
// (5) Determine # of dimes.
dimes = amountInCents / CENTS_PER_DIME;

// 6) Reset the amountInCents.
amountInCents %= CENTS_PER_DIME;

// 7) Determine # of nickels.
nickels = amountInCents / CENTS_PER_NICKEL;

// 8) Reset the amountInCents.
amountInCents %= CENTS_PER_NICKEL;

// 9) Determine # of pennies.
pennies = amountInCents;
// 10) Display originalAmount and qty of each coin.
System.out.println("Original amount in cents: " +
    originalAmountInCents);
System.out.println("# of quarters: " + quarters);
System.out.println("# of dimes: " + dimes);
System.out.println("# of nickels: " + nickels);
System.out.println("# of pennies: " + pennies);
} // end main()
} // end ChangeMaker
Use ++ or -- to increment / decrement

```c
int count = 1;

count++;       // count = 2;
++count;       // count = 3;
```

```c
int count = 1;

int result = ++count;  // count = 2, result = 2
result = count++;      // count = 3, result = 2
```

The increment and decrement operators are often used to increment and decrement a variable by one.

There are two versions of each, a pre and a post increment. It does not matter which one you use in isolation, but you should settle on using one or the other consistently in your code. This will create less confusion to the readers of your code.

Although the operators may be used in expressions, it is not advisable due to the confusion they create. See the second example above, where the variable result has a different value depending if the pre or post increment is used.
Recap

- Variables name memory locations.
- Data types dictate memory allocation.
- Objects are instances of classes.
- Scanner is used for input.
- System.out is used for output.
- Const variables don't change.
The Class
String
System.out.println("Enter a positive integer [1-100]: ");

String prompt = "Enter a positive integer [1-100]: ";

System.out.println(prompt);
String first = "Howdie";
String last = "Doodie";

String name = first + last;   // name = HowdieDoodie
name = first + " " + last;   // name = Howdie Doodie

String message = "";
message += "Amount = ";   // message = "Amount = 
message += 45.50;   // message = "Amount = 45.50"
message += "\nPlease pay up";   // message = "Amount = 45.50
            Please pay up"
Strings being objects do have methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>charAt(Index)</code></td>
<td>Returns the character in the receiving string, at Index (0 based)</td>
</tr>
<tr>
<td>length()</td>
<td>Returns the number of characters in the receiving string</td>
</tr>
<tr>
<td><code>indexOf(A_String)</code></td>
<td>Returns the index of the first occurrence of A_String within the receiving string. Returns -1 if not found.</td>
</tr>
<tr>
<td><code>lastIndexOf(A_String)</code></td>
<td>Returns the index of the last occurrence of A_String within the receiving string. Returns -1 if not found.</td>
</tr>
<tr>
<td><code>toLowerCase()</code></td>
<td>Returns a new string in all lowercase characters.</td>
</tr>
<tr>
<td><code>toUpperCase()</code></td>
<td>Returns a new string in all uppercase characters.</td>
</tr>
<tr>
<td><code>substring(Start)</code></td>
<td>Returns a new string starting at Start and extending to the end of the receiving string.</td>
</tr>
<tr>
<td><code>substring(Start, End)</code></td>
<td>Returns a new string starting at Start and ending at End - 1 of the receiving string.</td>
</tr>
<tr>
<td><code>trim()</code></td>
<td>Returns a new string with leading and trailing whitespace removed.</td>
</tr>
<tr>
<td><code>equals(Other_String)</code></td>
<td>Returns true if the receiving string is equal to Other_String, false otherwise.</td>
</tr>
</tbody>
</table>

This table is a small sample of the methods available for the String class. We will be using a lot of these methods as well as others in days to come.

Keep in mind that the value of the string cannot change, and this is done for efficiency purposes. Although the value cannot change, the string variable can be assigned to another value.
Strings are immutable

```java
String name = "Howdie";
name = "Powdie";
```

![Diagram showing the immutability of strings]
Here's your roadmap for escaping

```java
String title;
title = "I love "Java",
cause it\'s hot!";
System.out.println(title);
```

I love "Java",
cause it's hot!
Unicode
The new kid on the block

**ASCII** (older standard)
American Standard Code for Information Interchange
1 Byte per character
256 total characters

**UNICODE** (newer standard)
2 Bytes per characters
65,536 total characters
includes the ASCII set
Recap

- String literals are enclosed in double quotes.
- `+` concatenates two strings together.
- Strings are immutable.
- escape `\`, `\"` and `\\`
Keyboard and Screen IO
Output goes to the screen

```java
class obj
    method
        System.out.println("This is some text.");
```
import java.util.Scanner;

... 
Scanner keyboard = new Scanner(System.in);

int value = keyboard.nextInt(); // read next int
double value = keyboard.nextDouble(); // read next double
String value = keyboard.next(); // read next word
String value = keyboard.nextLine(); // read entire line
package scanneriodemo;

import java.util.Scanner;

public class ScannerIODemo {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);

        System.out.print("Enter two integers: ");
        int int1 = keyboard.nextInt();
        int int2 = keyboard.nextInt();

        System.out.println("You entered " + int1 + " and " + int2);
    }
}
Application Deconstructed
<ScannerIODemo.java>

```
System.out.print("Now enter two doubles: ");
double double1 = keyboard.nextDouble();
double double2 = keyboard.nextDouble();

System.out.println("You entered " + double1 + " and " +
double2);

System.out.print("Now enter two words: ");
String string1 = keyboard.next();
String string2 = keyboard.next(); // newline is still
in
keyboard.nextLine(); // so, let's remove
it

System.out.println("You entered " + string1 + " and " +
string2);
```
System.out.print("Now enter an entire sentence: ");
String sentence = keyboard.nextLine();

System.out.println("You entered " + sentence + "]");
} // end main()
} // end ScannerIODemo
Be wary of the next-nextLine combo

Given the following input:

```
1 is the
loneliest number.
```

```java
int num = keyboard.nextInt();
String str1 = keyboard.nextLine();
String str2 = keyboard.nextLine();
```

<table>
<thead>
<tr>
<th>num:</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>str1:</td>
<td>1 is the</td>
</tr>
<tr>
<td>str2:</td>
<td>loneliest number</td>
</tr>
</tbody>
</table>
Be wary of the next-nextLine combo

Given the following input:

1
is the
loneliest number.

```java
int num = keyboard.nextInt();
String str1 = keyboard.nextLine();
String str2 = keyboard.nextLine();

num: 1
str1: ""
str2: is the
```

Notice how str1 does not contain the string "is the", but instead the empty string ".". The reason for this is that the nextInt() method reads up to, but not including the newline character, which still remains to be read.

The next method is nextLine(), which reads up to and including the newline, but since there are no more characters to read before the newline, the empty string is returned instead.

The last nextLine() method reads the second line in its entirety into str1.

Basically, the next() method does not remove the delimiter from the input buffer, while the nextLine() does, so be careful with this duo sequence. To solve, add another nextLine() after the first next() to remove the remaining newline character.
The input delimiter can be changed

Scanner KeyboardComma = new Scanner(System.in);

keyboardComma.useDelimiter(",");
package delimiterdemo;
import java.util.Scanner;

public class DelimiterDemo {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        Scanner keyboardComma = new Scanner(System.in);

        keyboardComma.useDelimiter(",");
        System.out.print("Enter two words: ");

        String word1 = keyboard.next();
        String word2 = keyboard.nextLine();
        System.out.println("Word1 = "+word1 + ", Word2 = "+word2 + ");
    }
}
System.out.println();
System.out.print("Enter two words: ");

word1 = keyboardComma.next();
word2 = keyboardComma.nextLine();
System.out.println("Word1 = "+word1 + ", Word2 = "+word2 + ");

} // end main()
} // end DelimiterDemo
<table>
<thead>
<tr>
<th>Format Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%c</td>
<td>A single character.</td>
</tr>
<tr>
<td>%2c</td>
<td>A single character in a field of 2 spaces. Right aligned.</td>
</tr>
<tr>
<td>%d</td>
<td>An integer.</td>
</tr>
<tr>
<td>%5d</td>
<td>An integer in a field of 5 spaces. Right aligned.</td>
</tr>
<tr>
<td>%f</td>
<td>A floating point number.</td>
</tr>
<tr>
<td>%6.2f</td>
<td>A floating point number in a field of 6 spaces with 2 digits after the decimal. Right aligned.</td>
</tr>
<tr>
<td>%1.2f</td>
<td>A floating point number with 2 digits after the decimal in a field of as needed. Right aligned.</td>
</tr>
<tr>
<td>%e</td>
<td>A floating point number in exponential format.</td>
</tr>
<tr>
<td>%s</td>
<td>A string in a field as big as needed.</td>
</tr>
<tr>
<td>%10s</td>
<td>A string in a field of 10 spaces. Right aligned.</td>
</tr>
</tbody>
</table>
double price = 19.5;
int quantity = 2;
String item = "Widgets";

System.out.printf("%10s sold:%4d at $5.2f. Total = $1.2f", item, quantity, price, quantity * price);

"Widgets sold: 2 at $19.50. Total = $39.00"
Recap

- Output goes to the screen.
- Input comes from the keyboard.
- Heads up fo the next-nextLine debacle.
- Format your output.
Comments and Style

Καλώς ήρθατε
Variables should be self-documenting

```c
int count;       // count of what?
int countOfApples;

int sum = 0;     // sum of what?
int sumOfIntegerNumbers = 0;

double height;   // in what units?
double heightInInches;
```
Comments should explain your code

// Each line starts with a pair of // and can be
// placed anywhere. Everything after the //
// is ignored.

/*
   This second form is a bit more
   convenient for commenting a large
   section of code.
*/

/**
   Last, but certainly not least, this style
   allows for the use of javadoc, a utility
   that can generate html style help docs.
*/
Indentation adds structure to your code

// Without indentation.
public class Demo {
    public static void main(String[] args) {
        System.out.println("Ouch!");
    } // end main()
} // end Demo

// With indentation.
public class Demo {
    public static void main(String[] args) {
        System.out.println("Nice!");
    } // end main()
} // end Demo
Named constants enhance readability and maintainability

// Without named constants.
public class Demo {
    public static void main(String[] args) {
        double radiusInInches = 5.0;

        System.out.println("Circumference = " + 2 * 3.1415 * radiusInInches);
        System.out.println("Area = " + 3.1415 * radiusInInches * radiusInInches);
    } // end main()
} // end Demo

// With named constants.
public class Demo {
    public static final double PI = 3.1415;
    public static void main(String[] args) {
        double radiusInInches = 5.0;

        System.out.println("Circumference = " + 2 * PI * radiusInInches);
        System.out.println("Area = " + PI * radiusInInches * radiusInInches);
    } // end main()
} // end Demo
Recap

- Use lots of consts with your graphics.
- Use JFrame for GUI apps.
- Use JOptionPane for GUI flavored IO.