Chapter 3: Working with Data

Objectives

Students should

- Be familiar with the eight primitive data types and non-primitive data types.
- Understand memory allocations and value assignments of variables.
- Be able to use operators and some methods to do some computations.

Data Types in Java

There are 2 main types of data in Java.

1. Primitive data types
2. Classes

Primitive data types are data types considered basic to Java. They cannot be added or changed. There are eight primitive data types in Java including 4 types for integer values, 2 types for floating-point values, 1 type for characters, and 1 type for logical values (true/false).

Value type	Primitive data types
Integer (E.g.: 5, -1, 0, etc.)	`byte, short, int, long`
Floating-point (E.g.: 1.0, 9.75, -3.06, etc.)	`float, double`
Character (E.g.: 'a', 'ก', '@', '4', etc.)	`char`
Logical (true/false)	`boolean`

Table 1: Primitive data types in Java

Classes are more complex data types. There are classes that are standard to Java such as String, Rectangle, etc. Also, new classes can be defined by programmers when needed.

Primitive Data Type for Integers

There are four primitive data types for integer values, including `byte`, `short`, `int`, and `long`. They are different in the sizes of the space they occupied in memory. The reason why Java has multiple integer (as well as, floating-point) types is to allow programmer to access all the types support natively by various computer hardware and let the program works efficiently.

Recall that a space of 1 byte (8 bits) can store 2^8 different things.

A `byte` occupies 1 byte of memory. It can represent one of 2^8 (256) integers in the range -128, -127, …, 0, 1, …, 126, 127.

A `short` occupies 2 bytes of memory. It can represent one of 2^{16} (65,536) integers in the range -32,678, …, 0, 1, …, 32,767.

An `int` occupies 4 bytes of memory. It can represent one of 2^{32} (4,294,967,296) integers in the range -2,147,483,648, …, 0, 1, …, 2,147,483,647.

A `long` occupies 8 bytes of memory. It can represent one of 2^{64} integers in the range -2^{63}, …, 0, 1, …, $2^{64}-1$.

The default primitive type for integer value is `int`. Still, choosing to use suitable data types for your integer value leads to efficient data processing and memory usage.

Primitive Types for Floating-Point Values

There are two primitive data types for floating point values. They are `float`, and `double`. They are different in their sizes and precisions. The default primitive type for integer value is `double`.

A `float` occupies 4 bytes of memory. Its range is from -3.4×10^{38} to 3.4×10^{38}, with typical precision of 6-9 decimal points.

A `double` occupies 8 bytes of memory. Its range is from -1.8×10^{308} to 1.8×10^{308}, with typical precision of 15-17 decimal points.

There are two ways to write floating-point values. They can be written in either decimal or scientific notation. 123.45, 0.001, 1.0, 8.357, 1., and .9 are in decimal notation. 1.2345`E`2, 10`E`-4, 1`E`0, 0.8375`E`1, 1000`E`-3, and 9.0`E`-1 are in scientific notation representing 1.2345×10^{2}, 10×10^{-4}, 1×10^{0}, 0.8375×10^{1}, 1000×10^{-3}, and 9.0×10^{-1} respectively. The character `E` can also be replaced with `e`.

Primitive Type for Characters

The primitive data type for characters is `char`. A character is represented with single quotes. For example the character for the English alphabet Q is represented using 'Q'.

Characters include alphabets in different languages ('a', 'b', 'ก'), numbers in different languages ('1', '2', '๑'), symbols ('%', '#', '{'), and escape sequences ('\t', '\n', '\"');

In fact, the underlying representation of a `char` is an integer in the Unicode character set coding. Characters are encoded in 2 bytes using the Unicode character set coding.

Character	Unicode encoding
'A'	65
'B'	66
'a'	97
'b'	98
'1'	49
'#'	35

Table 2: Some examples of characters with their corresponding unicode encodings

Unicode character set encoding guarantees that '0' < '1' < '2' < … < '9', 'a' < 'b' < 'c' < … < 'z', and 'A' < 'B' < 'C' < … < 'Z'. Also,

'0'+1 is '1', '1'+2 is '3', …, '8'+1 is '9',

'a'+1 is 'b', 'b'+1 is 'c', …, 'y'+1 is 'z', and

'A'+1 is 'B', 'B'+1 is 'C', …, 'Y'+1 is 'Z'.

Primitive Type for Logical Values

There are only two different logical values, which are 'true' and 'false'. A primitive type called `boolean` is used for logical values. The size of this type is 1 bit.

In Java, 0 does not mean 'false' and 1 does not mean 'true'. Logical values and numeric values cannot be interchangeably.

String

String is an example of a data type that is not a primitive data type. String is a standard class in Java. It is used for representing a sequence of one or more characters. Double quotes are used to designate the String type. For example, "ISE" is a data of String type. Be aware that "100" is not the same as the integer 100, and "Q" is not the same as the character 'Q'.

Assigning Data to Variables

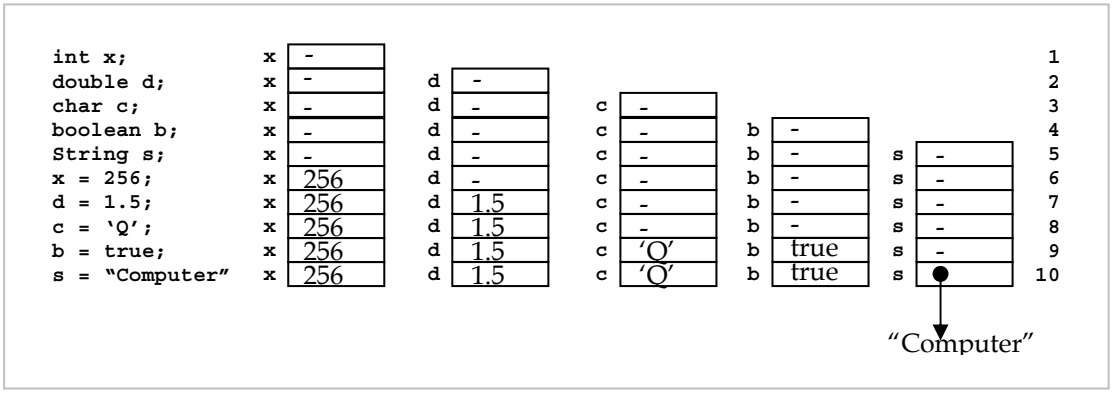
Data can be assigned to or stored in variables using the assignment operator (=). Like data, variables have types. A variable can only store a value that has the same data type as itself, unless the conversion can be done automatically (more on this later).

Variable types are specified when the variables are declared, using the name of the desired data types followed by the name of the variables. Do not forget semicolons at the end.

For each variable whose type is one of the primitive data types, memory space of the size corresponding to its type is allocated. The allocated space is used for storing the value of that type. However, it works differently for a variable of a non-primitive type. The memory space allocated for such a variable is for what is called a *reference*. When assigned with a value, the reference points, or refers, to the memory location that actually stores that value.

We have discussed briefly about how to declare variables and assign values to them in the last chapter. Here, we revisit it again with another example. This time, attention should be paid on the data types.

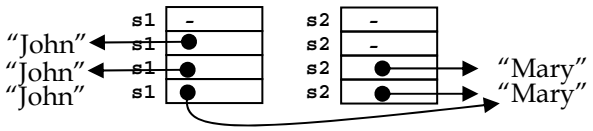
To better illustrate the memory allocation and value assignments of variables, we add to the following example code segment with some illustration of the allocated memory. Here, we represent a variable by a box captioned with its name. When a value is assigned to a variable, we write the value in side the box associated with that variable.



On the first four lines of the above code segment, a variable `x` of type `int`, a variable `d` of type `double`, a variable `c` of type `char`, and a variable `b` of type `boolean` are created and allocated with memory space of the size according to the type. On the fifth line, `String s` is the declaration of a variable `s` of String type. On this line, a reference to String is created but it has not referred to anything yet. Variables are assigned with values of the corresponding data types on the last five lines. For the last line, the reference in `s` is made to point to the String "Computer" located somewhere else in the memory.

The next example shows String assignment between two variables.

```
String s1, s2;
s1 = "John";
s2 = "Mary";
s1 = s2;
```



Here is another code segment showing what we can do.

```
int x,y = 8;
double z = 0.6, w;
double k=3.0;
w = k;
x = y;
```

The first two lines show how to declare two variables of the same type in one statement. Also, they show how to assign values to variables at the same time as the variables are declared. Note that, the integer value 8 is only assigned to `y`, not `x`, on the first line. On second line from the bottom, the `double` value stored in `k` is assigned to the variable `w`. On the last line, the `int` value stored in `y` is assigned to the variable `x`.

Here are some examples of bad variable declarations and/or assignments.

```
int i = 9                                          // no semicolon at the end
int j = 1.0;                                            // mismatch data type
boolean done = "false";                                 // mismatch data type
char input character = 'x';              // syntax error or illegal identifier
Int k = 1;                                             // Undefined data type
double k; m = 5e-13;                                  // undefined variable m
char class = 'A';                       // use a reserve word as an identifier
String s = 'W';                                         // mismatch data type
```

```
int i;
int k = 2;
int i = 6;                                  // declaration of redundant variable i
```

Final Variables

Sometimes, we want to store a value that cannot or will not be changed as long as the program has not terminated in a variable. The variable with such an unchangeable value is referred to as a *final variable*. The keyword `final` is used when a variable is declared so that the value of that variable cannot be changed once it is initialized, or assigned for the first time. Programmers usually use all-uppercase letters for the identifiers of final variables, and use underscore (_) to separate words in the identifiers (E.g.: YOUNG_S_MODULUS, SPEED_OF_LIGHT). Attempting to change the value of a final variable after a value has been assigned results in an error.

For example:

```
final double G = 6.67e10-11;
final double SPEED_OF_SOUND;
SPEED_OF_SOUND = 349.5; // Speed of sound at 30 degree celcius
```

The following code segment will produce an error due to the assignment in the last line.

```
final int C;
int k = 6;
C = 80;
C = k + 300;
```

Un-initialized Variables

When a variable is declared, a space in the memory is reserved for the size of the type of that variable. However, as long as it is not assigned with a value, the variable does not contain any meaningful value, and it is said that the variable has not been *intitialized*. If the value of an un-initialized variable is used, an error occurs.

The following code segment will produce an error due to the attempt to use an un-initialized variable.

```
int x, y = 2;
System.out.println(x+y);
```

Operators

Values can be manipulated using built-in arithmetic and logic operators (such as +, -, *, /, %), or using available methods (such as abs(), sqrt(), exp(), floor(), log(), getTime()). Many methods are defined in some standard classes such as the *Math* class. Although you are expected to be able to use the methods that have previously seen in this class, such as *print()* and *println()*, we will defer the detailed discussion on using methods for now. In this chapter, we will mostly discuss about using built-in operators and some small number of selected methods to manipulate data.

Arithmetic operators that you should know is addition (+), subtraction (-), multiplication (*), division (/), modulo (%), for which a%b returns the remainder if a÷b, and negation (-).

Logic operators that you should know is "logic and" (&&), "logic or" (||), and negation (!).

Portions of code that are either constant values, variables, methods, or any combinations of the mentioned items from which some values can be computed are called *expressions*.

Logic operator	Usage
&&	`a&&b` yields true if and only if both `a` and `b` are true.
||	`a||b` yields false if and only if both `a` and `b` are false.
!	`!a` yields the opposite logical value of `a`.

Table 3: Description of logic operators

Operators that require two operands are called *binary operators*, while operators that require only one operand are called *unary operators*.

Parentheses, (), are called *grouping operators*. They indicate the portions of the calculation that have to be done first.

Values can be compared using relational *equality operators*, == and !=, which return either one of the boolean values depending on the value of the operand. `a==b` yields true if and only if `a` and `b` have the same logical value. `a!=b` yields true if and only if `a` and `b` have different logical value. Comparison can also be done using <, >, <=, and >=.

Below are two examples showing how values of variables change due to value assignments and the use of some operators.

```
int i = 2, j, k;
j = 3;
k = i + j;
i = i + k;
```

	i	j	k
`int i = 2, j, k;`	2	-	-
`j = 3;`	2	3	-
`k = i + j;`	2	3	5
`i = i + k;`	8	3	5

```
boolean p = true;
boolean q = false, r = true, s;
s = p && q;
s = s || r;
p = (s == q);
```

	p	q	r	s
`boolean p = true;`	true			
`boolean q = false, r = true, s;`	true	false	true	-
`s = p && q;`	true	false	true	false
`s = s || r;`	true	false	true	true
`p = (s == q);`	false	false	true	true

String and the addition operator (+)

The addition operator (+) can be used to concatenate two Strings together. For example, "Comp"+"uter." will results in "Computer". The concatenation can also be used upon Strings that are stored in variables, such as:

```
String s1 = "Computer", s2 = "ized", s3;
s3 = s1 + s2;
```

The variable `s3` will contain "Computerized".

Whenever one of the operands of the addition operator is a String, the operator performs the String concatenation. Thus, if a String is added with values of different data types, the compiler will try to convert the values that are not String to String automatically. Good news is that the automatic conversion usually returns the String that makes very much sense! E.g.: numeric values will be converted to the Strings whose contents are consistent with the original numbers.

Observe this source code and its output.

```
public class StringConcat                                                    1
{                                                                            2
    public static void main(String[] args)                                  3
    {                                                                        4
        double distance, speed;                                              5
        distance = 2500; // meters                                           6
        speed = 80; // km per hour                                           7
        System.out.print("It would take a car running at "+speed+" km/h ");  8
        System.out.print((distance/1000/speed)*60*60+" sec. to travel ");    9
        System.out.println(distance/1000+" km.");                           10
    }                                                                       11
}                                                                           12
```

```
C:\WINDOWS\system32\cmd.exe
C:>javac StringConcat.java

C:>java StringConcat
It would take a car running at 80.0 km/h 112.5 sec. to travel 2.5 km.

C:>_
```

Some useful methods

Here are some example methods that provide useful mathematic functions.

`Math.abs(<a numeric value>);`
returns the absolute value of the input value.

`Math.round(<a numeric value>);`
returns the integer nearest to the input value.

`Math.ceil(<a numeric value>);`
returns the smallest integer that is bigger than or equal to the input value.

`Math.floor(<a numeric value>);`
returns the biggest integer that is smaller than or equal to the input value.

`Math.exp(<a numeric value>);`
returns the exponential of the input value.

`Math.max(<a numeric value>,<a numeric value>);`
returns the bigger between the two input values.

`Math.min(<a numeric value>,<a numeric value>);`
returns the smaller between the two input values.

`Math.pow(<a numeric value>,<a numeric value>);`
returns the value of the first value raised to the power of the second value.

`Math.sqrt(<a numeric value>);`
returns the square root of the input value.

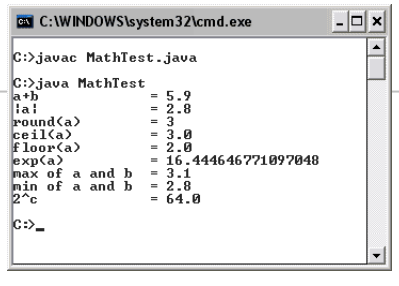
`Math.sin(<a numeric value >);`
returns the trigonometric sine value of the input value.

`Math.cos(<a numeric value >);`
returns the trigonometric cosine value of the input value.

`Math.tan(<a numeric value >);`
returns the trigonometric tangent value of the input value.

The source code below shows how these methods work.

```
public class MathTest                                                    1
{                                                                        2
    public static void main(String[] args)                              3
    {                                                                    4
        double a = 2.8, b = 3.1, c = 6.0;                                5
        System.out.println("a+b \t\t= " + (a+b));                        6
        System.out.println("|a| \t\t= " + Math.abs(a));                  7
        System.out.println("round(a) \t= " + Math.round(a));             8
        System.out.println("ceil(a) \t= " + Math.ceil(a));               9
        System.out.println("floor(a) \t= " + Math.floor(a));            10
        System.out.println("exp(a) \t\t= " + Math.exp(a));              11
        System.out.println("max of a and b \t= " + Math.max(a,b));      12
        System.out.println("min of a and b \t= " + Math.min(a,b));      13
        System.out.println("2^c \t\t= "+Math.pow(2,c));                 14
    }
}
```



Precedence and Associativity

Consider the following expression.

```
int myNumber = 3 + 2 * 6;
```

This expression is evaluated to 30 if the addition operator is executed first. However, it is 15 if the multiplication operator is executed first. In fact, Java compiler has no problem with such ambiguity. Order of the operators can be determined using *Precedence* and *association* rules.

Each operator is assigned a precedence level. Operators with higher precedence levels are executed before ones with lower precedence levels. Associativity is also assigned to operators with the same precedence level. It indicates whether operators to the left or to the right are to be executed first, in the case of equal precedence levels.

Expressions in parentheses () are executed first. In the case of nested parentheses, the expression in the innermost pair is executed first.

Operator	Precedence	Associativity
Grouping operator (())	17	None
Unary operator (+, -, !)	13	Right
Multiplicative operator (*, /, %)	12	Left
Additive operator (+, -)	11	Left
Relational ordering (<, >, <=, >=)	10	Left
Relational equality (==, !=)	9	Left
Logical and (&&)	4	Left
Logical or (||)	3	Left
Assignment (=)	1	Right

Table 4: Precedence and associativity of some basic operators

The following expression:

```
-9.0+5.0*3.0-1.0/0.5 >= 5.0%2.0&&6.0+3.0-9.0==0
```

is equivalent to:

```
((-9.0)+(5.0*3.0)-(1.0/0.5) >= (5.0%2.0))&&(((6.0+3.0)-9.0)==0)
((-9.0)+15.0      - 2.0       >=  1.0      )&&((     9.0 -9.0)==0)
(             4.0             >=  1.0      )&&(             0 ==0)
(                     true                 )&&(       true       )
true.
```

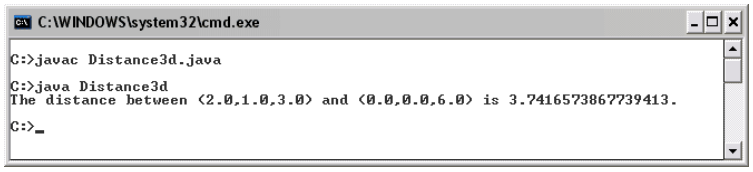
Examples

The distance, d, between two points in the three-dimensional space (x_1,y_1,z_1) and (x_2,y_2,z_2) can be computed from:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

The following program computes and shows the distance between (2,1,3) and (0,0,6).

```
public class Distance3d
{
    public static void main(String[] args)
    {
        double x1,y1,z1,x2,y2,z2, d;
        double xDiff, yDiff, zDiff;
        x1 = 2.0; y1 = 1.0; z1 = 3.0;
        x2 = 0.0; y2 = 0.0; z2 = 6.0;
        xDiff = x1-x2;
        yDiff = y1-y2;
        zDiff = z1-z2;
        d = Math.sqrt(xDiff*xDiff+yDiff*yDiff+zDiff*zDiff);
        System.out.print("The distance between ("+x1+","+y1+","+z1+") and");
        System.out.println(" ("+x2+","+y2+","+z2+") is "+d+".");
    }
}
```



Exercise

1. What are the differences between primitive data types and class?
2. How many different things can be represented using *n* bytes?
3. Which of the eight primitive data types can store a numeric value 2^{36}?
4. Give the `boolean` values of the following expressions.
    a. 01<2e-1
    b. 8+0.0 >= 8.0
    c. 'a'>'b'
    d. 2+3*2-6 == ((2+3)*2)-6
    e. 'a'>'$' | |'b'<'$'
    f. !(true | |'6'>'#')&&!false
5. Determine the resulting values of `x` and `y` in the following code.

```
public class  Ex3_5
{
    public static void main(String[] args)
    {
        int x=0,y=0;
        x = y + 1;
        y = x + 1;
    }
}
```

6. Give the reason why the following code will not be compiled successfully.

```
public class Ex3_6
{
    public static void main(String[] args)
    {
        int x, y, z =3;
        y = x;
        z = y;
    }
}
```

7. Write a Java program that calculates and shows these values on screen.
   a. (6+5)(7-3)
   b. $e^{\sqrt{\left|-(6.5)^2+(3.7)^2\right|}}$
   c. $\sqrt[3]{\sin(1.2)}$
   d. The floating point result of $\frac{8}{1+10}$
   e. The biggest integer that is less than 3.75
   f. The smallest integer that is bigger than 3.75
   g. $\sum_{i=0}^{3} i^{(i+1)}$
8. Write a Java program that shows the truth values of the following statements on screen for all possible combination of `p` and `q`. (i.e. (`true,true`), (`true,false`), (`false,true`), and (`false,false`))
   a. `p` and `q`
   b. `p` or `q`
   c. either `p` or `q`
   d. either `p` or its negation
9. Modify Distance3D.java so that the distance *d* is calculated from:

$$d = \frac{\sqrt{w_x^{\,2}(x_1 - x_2)^2 + w_y^{\,2}(y_1 - y_2)^2 + w_z^{\,2}(z_1 - z_2)^2}}{\sqrt{w_x^{\,2} + w_y^{\,2} + w_z^{\,2}}}$$

   where $w_x = 0.5$, $w_y = 0.3$, and $w_z = 0.2$.
10. Write a Java program that performs the following steps.
    a. Declare two `int` variables named `x` and `y`.
    b. Assign 3 to `x`.
    c. Assign twice the value of `x` to `y`.
    d. Interchange the value of `x` and `y` (without explicitly assign 3 to `y`).
    e. Print the values of both variables on screen.
11. Show a single Java statement that can perform both tasks in step b and c in the last problem.