PART 2: NUMBERS AND STRINGS

We introduce programs that carry out numerical calculations, and operations on character strings.

Appendix A of "Software Design with Java 2". Chapter 2 of *Computing Concepts with Java 2 Essentials by Cay Horstmann (CCJE).*

- Integers.
- Variables.
- Assignments.
- Floating-point numbers.
- Reading input using *ConsoleReader* class.
- Strings.

Integers

Java programs mostly work with *objects* (instances of classes). But also deal with simple values: integers and floating-point numbers (*primitive* data types).

Suppose you visit the USA, and you have in your pocket 8 pennies, no nickels, 5 dimes and 6 quarters. How much have you got altogether?

To work out answer, you need to know that a penny is a 1 cent coin, a nickel is worth 5 cents, a dime is worth 10 cents, and a quarter is worth a quarter of a dollar or 25 cents.

EXAMPLE 2.1

```
public class Change1
{  /* Calculate how much money you have (in cents)
      if you have 6 quarters, 5 dimes, no nickels,
      and 8 pennies. */

  public static void main(String[] args)
  {  System.out.print("You have ");
     System.out.print(6*25 + 5*10 + 0*5 + 8);
     System.out.println(" cents.");
  }
}
```

This program consists of single class containing single method called *main*.

main contains three statements. When it is run, these produce three parts of a single line of output:

- Statement 1: writes “You have ” (no new-line)
- Statement 2: writes “208” (no new-line)
- Statement 3: writes “ cents.” (plus new-line)

So we get:

```
You have 208 cents.
```

On the screen.

Arithmetic Expressions

In a Java program the following operators can be used in arithmetic expressions:

+	addition
-	subtraction (also used for -number)
*	multiplication
/	division
%	remainder

In Java numbers are either integers or floating-point numbers.

If value is written without decimal point, e.g. 4, is treated as an integer. If written with decimal point, e.g. 4.0, treated as floating-point number.

Always use integers if you can. Handled more quickly, require less memory, and there is no danger of round-off errors.

Arithmetic Expressions

Division operator, /, has two different meanings. If used with two integers then denotes integer division. E.g. 11 / 4 = 2, (-11) / 4 = -2. [Also called the *div* operator]

If used with two floating-point values, or one integer and one floating-point value, denotes normal division. E.g. 10.5 / 2 = 5.25.

Remainder operator is only used with two integers. E.g. 11%4 = 3 and (-11)%4 = -3. [Also called the *mod* operator]

If Java evaluates m/n or m%n where n is zero, an error will be given.

Java operators *, / and % bind more tightly than + and -. E.g. 3+4*5 means 3+(4*5), not (3+4)*5.

Arithmetic Expressions

In expression with several + and - operators in row, or several *, / and % operators in row, operators applied from left to right.

E.g. 3-4-5 has same meaning as (3-4)-5, not 3-(4-5). Similarly 3/4/5 means (3/4)/5, not 3/(4/5) (which will produce error).

Recommendation: always bracket explicitly in such cases to show intended meaning!

Variables

Used to save results of computations so we can use them later in program.

A *variable* is a name that denotes a value.

In following example variable *total* represents total amount of money in your pocket.

Variables should always have meaningful names unless they are loop indexes or very temporary values.

Variable names always begin with lowercase letter, use capitals to separate words within name: `taxRate`.

EXAMPLE 2.2

```
public class Change2
{  /* Calculate how much money you have in dollars
      and cents if you have 6 quarters, 5 dimes,
      no nickels, and 8 pennies. */

   public static void main(String[] args)
   {  int total = 6*25 + 5*10 + 0*5 + 8;
      System.out.print("You have ");
      System.out.print(total/100);
      System.out.print(" dollars and ");
      System.out.print(total%100);
      System.out.println(" cents.");
   }
}
```


Variable Declarations

The statement

```
int total = 6*25 + 5*10 + 0*5 + 8;
```

says: create a variable called *total* and set its value to result of expression $6 * 25 + 5 * 10 + 0 * 5 + 8$. (ie, to 208).

The word *int* means value stored in *total* is always an integer value.

This statement is called a *variable declaration* (or variable definition).

The statement

```
System.out.print(total/100);
```

says: output quotient when value denoted by total is divided by 100 (integer division). I.e. value 208/100 (= 2) is output.

The statement

```
System.out.print(total%100);
```

says: output remainder when *total* is divided by 100. I.e. output 208%100 (= 8).

This is what the program will display on screen:

```
You have 2 dollars and 8 cents.
```


Using Variables to Simplify Programs

In example 2, use of the variable saves work (programmer doesn't have to repeat $6 * 25 + 5 * 10 + 0 * 5 + 8$ expression, and Java only needs to calculate its value once).

Variables also used to store values of subexpressions of a complicated expression (eg, income tax calculation), then combined to give value of complete expression.

Another use of variables is to clearly label values that are being used, so that it is easier to understand what is going on. Can choose whatever name you like for a variable, as long as the name is not already used for something else. In particular, you can choose a descriptive name that indicates meaning of value.

Convention: name of a variable always starts with a lower case letter.

EXAMPLE 2.3

```
public class Change3
{  /* Calculate how much money you have in dollars
      and cents if you have 6 quarters, 5 dimes,
      no nickels, and 8 pennies. */

   public static void main(String[] args)
   {  int quarters = 6;
      int dimes = 5;
      int nickels = 0;
      int cents = 8;

      int total =
            quarters*25 + dimes*10 + nickels*5 + cents;

      System.out.print("You have ");
      System.out.print(total/100);
```



```
      System.out.print(" dollars and ");
      System.out.print(total%100);
      System.out.println(" cents.");
   }
}
```

Using Variables

This program produces same output as previous one. Variables *quarters*, *dimes*, etc simply intended to make program easier for someone to understand. In larger programs, very important to keep program code as simple and easy to understand as possible.

Idea of variables is familiar from school mathematics. However, variables in programming languages such as Java are slightly different to maths variables.

In Java, when variable is created it is allocated an area in memory where its value is stored. Amount of memory allocated depends on type of value stated in declaration.

Variables of type int, like *total* above, will be allocated 32 bits of memory.

Variable Values

The integer value will be stored in fixed-point binary notation – but programmer rarely needs to consider this. However, size of storage of variable imposes limit on size of its value.

32-bit storage means value of *int* variable must lie in range -2147483648 to 2147483647. If you want bigger integers, replace *int* by *long*. Then variable will be given 64 bits, and can be up to 9223372036854775807.

There are also shorter integer variables of type short (16 bits) and byte (8 bits).

Assignments

Each variable has a name and a piece of memory. We call value held in the piece of memory the value *assigned* to the variable.

Unlike in maths this value does not have to remain the same. It can be replaced by another value while program is running.

Program starts by creating two variables called *stones* and *pounds* set to someone's weight in stones and pounds. (1 stone = 14 pounds.) Then obeys statement

```
pounds = 14*stones + pounds;
```

This says: calculate the value of expression $14 * stones + pounds$, and store result in variable *pounds*.

Ie, it replaces value originally stored in *pounds* (= 8) by the person's total weight in pounds (= 148).

EXAMPLE 2.4

```
public class Weight1
{  /* Calculate someone's weight in kilos
      given their weight in stones
      and pounds. */

   public static void main(String[] args)
   {  int stones = 10;
      int pounds = 8;   /* Initialisation of pounds */

      pounds = 14*stones + pounds; /* Update pounds */
      double kilos = pounds * 0.4536;
      System.out.print("You weigh ");
      System.out.print(kilos);
      System.out.println(" kilos.");
   }
}
```

The statement

```
double kilos = pounds * 0.4536;
```

is first example of use of a floating-point value. It says: create a variable called *kilos* of type *double*, and assign it value $pounds * 0.4536$. A variable of type *double* holds a double length floating point number, in 64 bits of memory.

Product of an integer and a floating-point number is calculated in floating-point. In this case answer will be 67.1328. Last three statements output answer:

```
You weigh 67.1328 kilos.
```


Assignment Statements

The statement

```
pounds = 14*stones + pounds;
```

is called an *assignment statement*. It does not create a new variable. It changes the value of an existing variable.

Whenever Java obeys an assignment statement it carries out two steps: first it evaluates expression on right of '=', then stores result in variable on left of '='.

Here it evaluates sum of $14 * stones$ plus original value in *pounds*, and then puts answer in *pounds*, overwriting old value. In effect, it adds $14 * stones$ to value of *pounds*.

Shorthand for this kind of assignment is:

```
pounds += 14*stones;
```

Increment/Decrement Statements

One of commonest examples of this kind of statement is when we want to add 1 to value of a variable:

```
x += 1;
```

However, there is an even shorter abbreviation:

```
x++;
```

Similarly, to subtract 1 from `x`, can write `x--`.

You can change value stored in a variable, but cannot change type of the variable.

For example, given declaration of *int* variable *pounds*, any new value assigned to it must be an integer, and will always be stored in same 32 bits of memory.

Initialisation

Possible to declare variable without specifying its initial value. For example:

```
int total;
```

will create variable *total* of type *int*. The value will be set later when an assignment statement is obeyed. In meantime no way of knowing what value *total* contains. It may have some arbitrary set of bits left in its 32-bit piece of memory from some previous program. We say the variable's value is 'undefined'.

Recommendation: explicitly initialise all variables, even if only with default values (like 0 for integers)!

Summary

We have seen these forms of statement:

- declaration statement (with an initial value):

  ```
  TYPE  NAME  =  EXPRESSION;
  ```

  For example, *TYPE* might be *int*, *NAME* might be *quarters*, *EXPRESSION* might be 6. Therefore

  ```
  int quarters = 6;
  ```

  is a declaration statement.

- declaration statement (without an initial value)

  ```
  TYPE  NAME;
  ```

- assignment statement

  ```
  NAME  =  EXPRESSION;
  ```

Java also allows declaration statements that introduce several new variables.

Following creates variables w, x, y and z, and gives initial values to x and z.

```
int w, x = 1, y, z = 2;
```

Not a good idea as hard to work out what is going on!

Constants

Used to hold values which are read but not changed by a program, for example

```
final int DAYS_PER_YEAR = 365;

System.out.println("Number of days in " + n + " years is: " +
                                DAYS_PER_YEAR*n);
```

final keyword means assignments to *DAYS_PER_YEAR* are invalid.

Constants usually written all in upper case, with "_" as word separator.

Floating-point Values

Name 'floating-point number' comes from way these numbers are represented in computer memory. Notation used similar to scientific notation (e.g. $1.075 * 10^{24}$), but uses base 2 instead of base 10. Discussed in detail in systems course units.

In Java, data type *double* usually used for floating-point values. 64 bits are used for a *double* value. Allows a very wide range of values: approximately $+/- 1.8 * 10^{308}$, with accuracy to 15 significant digits. *double* is short for 'double precision floating-point'.

Java has second data type for handling floating-point numbers, called *float*. This uses only 32 bits for values, so more economical in use of memory, but has only half precision of *double*.

A literal floating-point number can be written with a decimal point, e.g. 14.56 or 14.0. Or written with an exponent, e.g. 14.56E-12. Equivalent to $14.56 * 10^{-12}$.

Floating-point Variables

Can declare a variable to be of type *double*. For example

```
double ratio = 1.67;
```

Creates variable called *ratio*, allocates it 64-bit location in memory containing initial value 1.67.

Java provides methods that compute wide range of mathematical functions using double values. Here are some of them. They are all static methods (i.e. methods that do not belong to an object) in Java library class *Math*. Whenever we use a static method that belongs to a library class, we must prefix its name with name of the class it belongs to:

Math.sin(x)	sine of x
Math.cos(x)	cosine of x
Math.tan(x)	tangent of x
Math.exp(x)	e^x
Math.log(x)	natural logarithm of x
Math.abs(x)	absolute value of x
Math.floor(x)	largest integer $\leq$ x
Math.ceil(x)	smallest integer $\geq$ x

In each case parameter x has type *double* and result is of type *double*.

Numeric Type Conversions

Java will accept *int* value being used where a *double* value is needed. Will convert integer into corresponding floating-point value. For example, in:

```
int m = 3;
int n = 4;
double x = m*n;
```

$m * n$ is evaluated using integer arithmetic to give 32-bit integer value 12. Then converted to 64-bit floating-point value 12.0, which is then stored in x.

If third statement is replaced by

```
double x = m/n;
```

x will be set to 0, because integer division will be used to evaluate m/n.

Casts

To force floating-point division to be used to work out value of x, can write:

```
double x = ((double) m) / n;
```

The term `(double)` is called a *cast.* By putting cast in front of m, we are asking Java to convert value of m to a double value, i.e. a 64-bit floating-point value. Once it has done this, will apply the / operator with one *double* value and one int *value.* So get floating-point division.

To convert floating-point numbers to integers, two approaches:

1. Take integer part of floating-point number. Eg, floating-point number 4.7 becomes integer 4. In Java do this by an int cast:

   ```
   double x = 4.7;
   int i = (int) x;
   ```

   Second statement converts `x` to integer 4, then stores it in `i`. If `x` had been set to -4.7, `i` would be assigned -4.

2. Alternatively, take the nearest integer. This is called rounding the number. For example, nearest integer to 4.7 is 5. Following statement rounds value of x and stores it in i, provided value is not negative:

   ```
   int i = (int) (x + 0.5);
     // Set i to nearest int to x (for x >= 0)
   ```

   To assign *int* variable i to nearest integer to x, where x may be positive or negative, use $Math.round(x)$. Produces a *long*. To assign result to i (assuming answer fits in an *int* variable) apply `(int)` cast to it:

   ```
   int i = (int) (Math.round(x));
   ```


Reading Input Using the ConsoleReader Class

Example programs we've seen all carry out simple calculations using values coded into program. To change values would have to alter the programs.

Instead programs can obtain initial values by reading what user types on keyboard.

Programs display information on screen using *print* and *println* methods of object *System.out*. This object has access to screen, its methods transfer string and number data to the screen.

Similarly for input need an object connected to keyboard, with methods to access numbers and strings of characters that user types.

Reading Input Using the ConsoleReader Class

In Sections 2.7, 2.8 and 3.10 of CCJE, Cay Horstmann describes how to do this. *ConsoleReader* packages up such access methods.

To create an object for reading from keyboard, insert following statement in your program:

```
ConsoleReader in = new ConsoleReader(System.in);
```

Creates new *ConsoleReader* object to access *System.in* object which is connected to keyboard.

The *ConsoleReader.java* file must be in the same directory as your own program file.

Using ConsoleReader

Main methods:

in.readInt()	This returns an *int* value. E.g. statement `int dimes = in.readInt();` creates variable *dimes* and gives it next value typed by user.
in.readDouble()	This returns a *double* value typed by the user.
in.readLine()	This returns a string, the next line of text typed by user.

Exceptions occur if attempt to read non-int string as int, etc.

Using ConsoleReader

Here is new version of weight conversion program.

EXAMPLE 2.5

```
public class Weight2
{  /* Read user's weight in stones and pounds,
      and output their weight in kilos. */

   public static void main(String[] args)
   {  ConsoleReader in =
          new ConsoleReader(System.in);

      System.out.println("Tell me your weight.");
      System.out.print("How many stones: ");
      int stones = in.readInt();
      System.out.print("How many pounds: ");
      int pounds = in.readInt();
```



```
      pounds = 14*stones + pounds;
      double kilos = pounds * 0.4536;
      System.out.print("You weigh ");
      System.out.print(kilos);
      System.out.println(" kilos.");
   }
}
```

Here is example of what might appear on screen when program is run. (User's input in italic.)

```
Tell me your weight.
How many stones:  8
How many pounds:  5
You weigh 53.712 kilos.
```

Strings

String values in Java are written between double quotes:

`"the dog"`

Can also have string with no characters. The empty string: `""`.

Control characters such as tabs and newlines are written in strings using \, as are literal quotes.

The following control characters can be used in strings:

\n	newline
\t	tab
\b	backspace
\r	return (i.e. go back to start of the line)
\f	line feed (i.e. move to next line without going to start)
\\	\ character
\'	' character
\"	" character

For example, the statement

```
System.out.print("The\nend");
```

will write 'The' on one line, followed by 'end' on the next.

Variables of type String

Can define variables of type String:

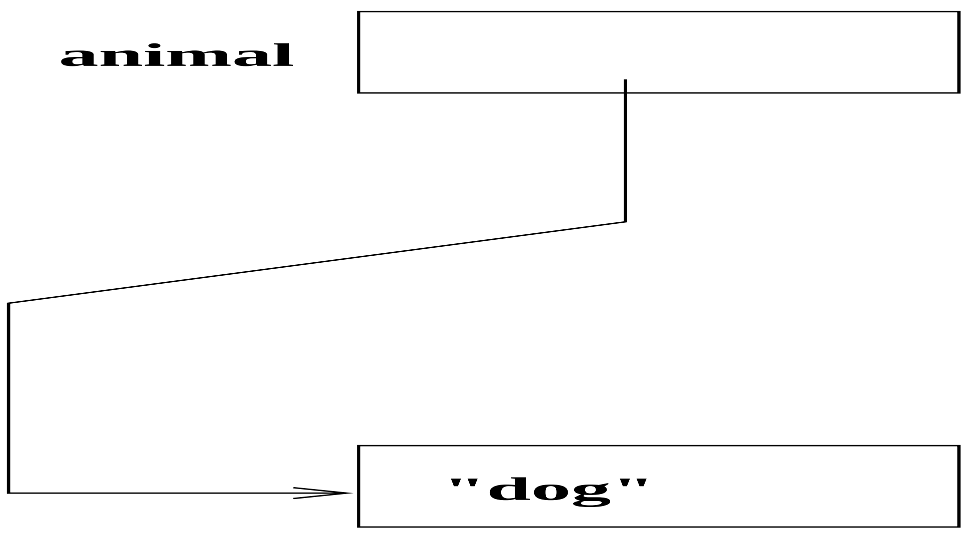
```
String animal = "dog";
```

Because strings treated like objects in Java, *animal* will actually store a memory address – address where string object for `"dog"` is stored.

Above declaration does:

1. creates variable called *animal*
2. string `"dog"` is stored in *another* part of memory.
3. reference of place where string has been stored is assigned to *animal*.

Memory Arrangement for String Variables



String Operators

Very frequently used operator is *concatenation*, denoted by '+'. If $st1$ and $st2$ are strings, then $st1 + st2$ denotes string consisting of all characters of $st1$ followed by all characters of $st2$.

For example, if variable *animal* has been assigned `"dog"`, then expression

```
"The " + animal + " ran away."
```

denotes the string

```
"The dog ran away."
```

So the statement

```
System.out.println("The " + animal + " ran away.");
```

displays message `"The dog ran away."` on screen.

String Operators

Java allows you to combine numbers and strings using the + operator. For example, if *howMany* is integer variable with value 10, then expression

```
"The " + animal + " ran away " + howMany + " times".
```

denotes the string

```
"The dog ran away 10 times."
```

Java converts value of *howMany* into string of digits used to represent value in standard decimal notation.

Could have used this in examples 2 and 3. The code

```
System.out.print("You have ");
System.out.print(total/100);
System.out.print(" dollars and ");
System.out.print(total%100);
System.out.println(" cents.");
```

could be replaced by:

```
System.out.println("You have " +
                   (total/100) + " dollars and " +
                   (total%100) + " cents.");
```


Simple Interaction with Users

EXAMPLE 2.6

```
public class Chat
{ /* Chat to the user. */
  public static void main(String[] args)
  { ConsoleReader user =
      new ConsoleReader(System.in);

    System.out.println("Hello.  What is your name?");
    String name = user.readLine();
    System.out.println("How old are you " + name + "?");
    int age = user.readInt();
    System.out.print(age + " is a nice age, ");
    System.out.println("but " + (age+1) +
                       " is better.");
    System.out.println("See you later " + name + ".");
  }
```

```
}
```

Here is what might appear on screen when this program is run.

```
Hello.  What is your name?
```
Alice Brown
```
How old are you Alice Brown?
```
20
```
20 is a nice age, but 21 is better.
See you later Alice Brown.
```


String Methods

substring method. Java uses convention that characters in a string are numbered 0, 1, 2, ... starting at lefthand end.

If st denotes a string, m and n denote int values, the expression

```
st.substring(m,n)
```

returns string made up of characters of st, starting at position m and ending at position $n - 1$.

If variable *animal* has been assigned `elephant`, then expression

```
animal.substring(5,8)
```

returns the string `"ant"`:

```
e l e p h a n t
          | | |
0 1 2 3 4 5 6 7 8
```

String Methods

Some other standard methods to use with string *st*:

st.length()	Returns number of characters in *st*. E.g. `"Alice Brown".length()` returns value 11.
st.toLowerCase()	Returns lowercase version of *st* E.g. `"Alice Brown".toLowerCase()` returns `"alice brown"`.
st.toUpperCase()	Returns uppercase version of *st* E.g. `"Alice Brown".toUpperCase()` returns `"ALICE BROWN"`.

See *MakePassword* program, page 79 of CCJE (p71 in old edition).

Converting between numbers and strings

To convert number to string, combine number and empty string using + operator. Eg, if variable *result* contains value 142, then expression `"" + result` returns string `"142"`.

To convert string to number, use:

- `Integer.parseInt(st)` if string *st* represents an integer value, this will produce the value.
- `Double.parseDouble(st)` if string *st* represents a floating-point value, this will produce the value.

Consider following two expressions.

`"The result is " + 3 + 7`

and

`3 + 7 + " is the result"`

Seem to say same thing. But if you try outputting them using *System.out.println* will find results quite different.

To work out meaning of expression of several strings and numbers joined by + operators, use following rule.

To evaluate x+y where x and y are either numbers or strings, Java will:

1. If x and y are both strings, concatenates them.
2. If one of x and y is a string and other is a number, will convert number to string and concatenate with the other.
3. If x and y are both numbers, will add them.

4. x+y+z is evaluated as (x+y)+z.

So the expression

`"The result is " + 3 + 7`

is evaluated as `"The result is " + 3` first – obtaining `"The result is 3"`. Then evaluates `"The result is 3"` + 7 to give:

`"The result is 37"`

Next consider

`3 + 7 + " is the result"`

This time Java will first add 3 and 7 to give 10. Then evaluates 10 + `" is the result"` to give

`"10 is the result"`

How to get a Java CD-ROM

First you must obtain a blank CD-ROM. This can be bought from Department's Support Group, in 7D. However, they tell me that blank CD-ROMs can be obtained more cheaply from W.H.Smith's. (The nearest W.H.Smith's is at the bottom of Kingsway.)

When you have a blank CD-ROM, take it to room 7D and find the computer that will burn data onto CD-ROMs. It is the computer next to the printer. When it is free, place your blank CD-ROM in the reader, label uppermost as usual. Then follow the instructions on the screen. When you are asked to 'type the name of the CD', enter 'jdk13'. If in doubt, ask a demonstrator for help.

Exercises 2, hand-in: Friday 18th October, 4pm. Coursework office

1 Write, compile and run a program which prints out the values 1!, 2!, 3!, 4! and 5! on successive lines, ie, the values 1, 2, 3*2, 4*3*2 and 5*4*3*2. Do this by declaring a variable *int* $f\ =\ 1$; which is then multiplied by 2, 3, 4, 5 in turn. Don't use a loop.

2 Add a variable *int* $p\ =\ 1$; to the program of part 1, and print out the successive values $n!/2^n$ for $n = 1, \ldots, 5$. Do this by multiplying p by 2 each time f is multiplied, and then printing f/p.

3 Change p to be a `double` variable and write down the results you get.