Basic Computation (Savitch, Chapter 2)

TOPICS

- Variables and Data Types
- Expressions and Operators
- Integers and Real Numbers
- Characters and Strings
- Input and Output

Variables in a programming language

- Variables store information
  - You can think of them like boxes
  - They “hold” values
  - The value of a variable is its current contents

- Note that this differs from variable in math
  - In math, a variable is an “unknown”
    - Solving an equation reveals its value
  - In programming, variables are “place holders” for values
    - Their current value is always known
    - The program changes their values to achieve a goal

Data Types

- Variables are like boxes: they hold values
  - But you can’t put an elephant in a shoe box
  - Different boxes hold different types of things

- Therefore, variables have *data types*
  - The data type describes the set of values a variable might contain
  - The value of a variable is a member of the set defined by its data type
  - Examples: int, char, double, boolean, String
  - Java is a strongly typed language – only values of the correct type can be put into a variable

Creating Variables

- You create new variables through *declarations.*
  - Examples:
    - `int daysPerYear;`
    - `char vowel;`

- You assign values to variables using the *assignment operator* `=`.
  - Examples:
    - `daysPerYear = 365;`
    - `vowel = ‘a’;`

- Declaration and assignment can be combined.
  - Examples:
    - `int price = 25;`
    - `char c = ‘&’;`
More about Variables

- An uninitialized variable is useless
  - So it's good practice to initialize when declaring variables
  - can be done with one statement:
    ```java
    int daysPerYear = 365;
    ```
- Variables can be re-used:
  ```java
  int daysPerYear = 365;
  // random code here
  daysPerYear = 110;
  ```

Literals

- **Literals** are values that are directly recognized by Java:
  - numbers
    - 237, 10, 9, 1.5, 5.8, 99.999
  - characters
    - 'a', 'Z', '0', '$'
  - Strings
    - "hello", "there"

Java Identifiers

- An *identifier* is a name, such as the name of a variable.
- Identifiers may contain only
  - Letters
  - Digits (0 through 9)
  - The underscore character (_)
  - And the dollar sign symbol ($) which has a special meaning
- The first character cannot be a digit.

- Identifiers may not contain any spaces, dots (.), asterisks (*), or other characters:
  - 7-11 netscape.com util.* // not allowed!
- Identifiers can be arbitrarily long.
- Since Java is case sensitive, *stuff*, Stuff, and **STUFF** are different identifiers.
Keywords or Reserved Words

- Words such as if are called *keywords* or reserved words and have special, predefined meanings.
  - Cannot be used as identifiers.
  - See Appendix 1 for a complete list of Java keywords.
- Examples: `int`, `public`, `class`

Naming Conventions

- Class types begin with an uppercase letter (e.g. String).
- Primitive types begin with a lowercase letter (e.g. int).
- Variables of both class and primitive types begin with a lowercase letters (e.g. myName, myBalance).
- Multiword names are "punctuated" using uppercase letters.

Where to Declare Variables

- Declare a variable
  - Just before it is used or
  - At the beginning of the section of your program that is enclosed in `{}`:
    ```java
    public static void main(String[] args)
    {
      /* declare variables here */
      . . .
      /* code starts here */
      . . .
    }
    ```

Java Types

- In Java, there are two kinds of data types:
  - Primitive data types
    - Hold a single, indivisible piece of data
    - Pre-defined by the language
    - Examples: int, char, double, boolean
  - Reference data types (Classes)
    - Hold complex combinations of data
    - Programs may define new classes
    - Examples: String, Scanner, System
### Primitive Types

- **Integer types**: byte, short, int, and long
  - `int` is most common
- **Floating-point types**: float and double
  - `double` is more common
- **Character type**: char
- **Boolean type**: boolean

### Primitive Data Types

- The 4 most common primitive data types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>integer values</td>
<td>5</td>
</tr>
<tr>
<td>double</td>
<td>floating-point values</td>
<td>3.14</td>
</tr>
<tr>
<td>char</td>
<td>characters</td>
<td>‘J’</td>
</tr>
<tr>
<td>boolean</td>
<td>either true or false</td>
<td><code>true</code></td>
</tr>
</tbody>
</table>

Notice – there are no quotes around `true`!

### Assignment Statements

- An assignment statement is used to assign a value to a variable.
  ```java
  answer = 42;
  ```
- The "equal sign" is called the **assignment operator**.
- We say, "The variable named `answer` is assigned a value of 42," or more simply, "`answer` is assigned 42."
- General rule: `Var = expression;`
Building expressions: operators

- **Operators** can act on numbers and primitive data types
  - ‘+’ adds two numbers
  - ‘*’ multiplies two numbers
  - ‘-’ subtracts two numbers
  - ‘/’ divides two numbers
  - ‘==’ tests for equality and returns a Boolean

- Operators can also act on other expressions – build arbitrarily complicated expressions (just like math)

Expressions

- A program is a sequence of statements
  - Well, it also needs a header
  - But the program body lists statements
- Simplest statement is an assignment
- A simple expression looks like:
  \[ var1 \ op \ var2; \]
  - Where ‘var1’ and ‘var2’ are variables
  - ‘op’ is any operator (e.g. +, -, *)

Variations on Expressions

- Variables can be re-used across and even within expressions:
  
  ```
  int x = 7;
  x = x + 1;
  x = x + x;
  ```
  - Right hand side is **evaluated**
  - ... and the result is **assigned** to the lhs variable

More variations on expressions

- The **right hand side** of an assignment can be any mathematical expression:
  
  ```
  int y = x + (2 * z);
  ```
- When more than one operator appears
  - Parenthesis disambiguate
    - See above
  - Without parentheses, operator precedence rules apply
    - e.g., multiply before add, left before right
    - Better to rely on parentheses
Integers

- Numbers without fractional parts
  3, 47, -12
- Variables store integers with an assignment statement
  ```java
  int size = 7;
  ```
- Integer variables may be used like integer literals (i.e., number), e.g.,
  ```java
  size = size + 1;
  ```

Integer Arithmetic Operations

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Example</th>
<th>Evaluates to</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>45 + 5</td>
<td>50</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>657 – 57</td>
<td>600</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>7000 * 3</td>
<td>21000</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>13 / 7</td>
<td>1</td>
</tr>
<tr>
<td>%</td>
<td>Remainder</td>
<td>13 % 7</td>
<td>6</td>
</tr>
</tbody>
</table>

Remainder reminder: % and /

- For any pair of integers, x and y, there always exist two unique integers, q and r, such that $x = qy + r$ and $0 \leq r < y$
- q is called the quotient, r is the remainder when x is divided by y
- The operators, % and / compute them:
  - $r = x \% y$ and $q = x / y$
- Java isn’t mathematically correct for negative numbers

```java
int i = 10/3;

int j = 10 % 3;

int k = 13 % 5 / 2;
```
Additional Integer Operators

- **Self-assignment**
  
  ```java
  int temperature = 32;
  temperature = temperature + 10;
  ```

  What is temperature? 42

- **Increment**
  
  ```java
  cent++;  // equivalent to cent = cent + 1;
  ```

- **Decrement**
  
  ```java
  cent--;  // equivalent to cent = cent - 1;
  ```

Specialized Assignment Operators

- **Assignment operators can be combined with arithmetic operators including -, *, /, %.*
  
  ```java
  amount = amount + 5;
  ```

  can be written as

  ```java
  amount += 5;
  ```

  yielding the same results.

Parentheses and Precedence

- Parentheses can communicate the order in which arithmetic operations are performed

- examples:

  ```java
  (cost + tax) * discount
  ```

  ```java
  cost + (tax * discount)
  ```

- Without parentheses, an expression is evaluated according to the **rules of precedence**.

Precedence Rules

- **Figure 2.2 Precedence Rules**

  ```plaintext
  Highest Precedence
  First: the unary operators +, -, !, ++, and --
  Second: the binary arithmetic operators *, /, and %
  Third: the binary arithmetic operators + and -
  Lowest Precedence
  ```
Precedence Rules

- The *binary* arithmetic operators *, /, and %, have lower precedence than the *unary* operators +, −, ++, --, and !, but have higher precedence than the binary arithmetic operators + and −.

- When binary operators have equal precedence, the operator on the left acts before the operator(s) on the right.

Sample Expressions

<table>
<thead>
<tr>
<th>Ordinary Math</th>
<th>Java (Preferred Form)</th>
<th>Java (Parenthesized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rate + delta</td>
<td>rate * rate + delta</td>
<td>(rate * rate) + delta</td>
</tr>
<tr>
<td>2(salary + bonus)</td>
<td>2 * (salary + bonus)</td>
<td>2 * (salary + bonus)</td>
</tr>
<tr>
<td>1/time + 3 * mass</td>
<td>1 / (time + 3 * mass)</td>
<td>1 / (time + (3 * mass))</td>
</tr>
<tr>
<td>a – 7</td>
<td>(a – 7) / (t + 9 * v)</td>
<td>(a – 7) / (t + (9 * v))</td>
</tr>
<tr>
<td>t + 9.5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Real Numbers

- Also called floating-point numbers
- Numbers with fractional parts
  3.14159, 7.12, 9.0, 0.5e001, -16.3e+002

- Declared using the data type double
  
double pricePerPound = 3.99,
  taxRate = 0.05,
  shippingCost = 5.55;
  double pctProfit = 12.997;

double Arithmetic Operations

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>45.0 + 5.30</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>657.0 – 5.7</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>70.0 * 3.0</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>96.0 / 2.0</td>
</tr>
</tbody>
</table>
Numbers in Java

- *int* is of fixed size; a value that is too large to be stored in an *int* variable will not match the mathematical value.
- Example:

  ```java
  int x = 100000 * 100000;
  out.println( x );
  ```

  Will print: 1410065408

  Numerical overflow!
  No warning messages!

- It is not always possible to test *double* expressions for equality and obtain a correct result because of rounding errors (called “floating point error”).

  ```java
  public class ProblemDoublePrecision
  {
    public static void main( String[ ] args )
    {
      double val = 1.0/5.0+1.0/5.0+1.0/5.0-0.6;
      System.out.println( val );
    }
  }
  ```

  Should be zero!

How should you handle “floating point error”?
- Test to see if the value is within a margin of error

  ```java
  public class CheckDoubleEquality
  {
    public static void main( String[ ] args )
    {
      double val = 1.0/5.0+1.0/5.0+1.0/5.0-0.6;
      if ( Math.abs(val) < 0.00001 )
      {
        val = 0;
        System.out.println( val );
      }
    }
  }
  ```

Assignment Compatibilities

- Java is strongly typed.
  - You can’t, for example, assign a floating point value to a variable declared to store an integer.
  - But for convenience, some conversions between numbers are possible …
    ```java
doubleVariable = 7;
```  
  - … is possible even if doubleVariable is of type double, for example.
Assignment Compatibilities

- A value of each following type can be assigned to a variable of type to the right:
  - byte → short → int → long → float → double
    - but not to a variable of any type to the left.
- You can assign a value of type char to a variable of type int.
- … except through type casting

Type Casting

- A type cast changes the value of a variable from the declared type to some other type.
- For example,
  - double distance;
  - distance = 9.0;
  - int points;
  - points = (int)distance;
- Illegal without (int)

Type Casting

- The value of (int)distance is 9,
- The value of distance, both before and after the cast, is 9.0.
- The fractional part (to the right of the decimal point) is truncated rather than rounded.

Mixing Numeric Data Types

- **Widening conversion.** Java will automatically convert int expressions to double values without loss of information
  - int i = 5;
  - double x = i + 10.5;
  - double y = i;

- **Narowing conversion.** To convert double expressions to int requires a typecasting operation and truncation will occur
  - i = (int) (10.3 * 2);
  - To round-up instead of truncating add 0.5
  - i = (int) (10.3 * x + 0.5);
Characters

- Any key you type on the keyboard generates a character which may or may not be displayed on the screen (e.g., nonprinting characters)
- Characters are a primitive type in Java and are not equivalent to Strings
- Examples
  ```java
  char vitamin = 'A',
  chromosome = 'y',
  middleInitial = 'N';
  ```

Important Literal Characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A', ... , 'Z'</td>
<td>Uppercase letters</td>
</tr>
<tr>
<td>'a', ... , 'z'</td>
<td>Lowercase letters</td>
</tr>
<tr>
<td>'0', ... , '9'</td>
<td>Digits</td>
</tr>
<tr>
<td>'.', ',', '!', '&quot;', etc.</td>
<td>Punctuation Marks</td>
</tr>
<tr>
<td>' '</td>
<td>Blank</td>
</tr>
</tbody>
</table>
| '
' | New line |
| '\t' | Tab |
| '\' | Backslash |
| '\' | Single Right Quote |

The other meta-type: Classes

- A primitive data type is indivisible
  - They have no meaningful subparts
  - The primitives are defined by the language
    - int, char, double, etc.
- A class is a data type that contains many bits of information
  - For example, Strings (many primitive chars)
  - Many classes defined by the language
    - You can also define new ones...

Classes

- Classes have data and methods
  - The data may be primitives or collections of primitives, other even classes.
  - Methods are used instead of operators
- The period ('.') accesses methods of a class:
  ```java
  String greeting = "hello";
  char c = greeting.charAt(0);
  // c now equals 'h'
  ```
More about Strings

- **String** is defined in the `java.lang` package.
  - The `java.lang` package is automatically included in all programs, so you do not need to import it.
- String literals are defined in double-quotes;
  - Examples:
    ```java
    String t1 = "To be ";
    String t2 = "or not to be";
    System.out.println(t1.concat(t2)); // prints To be or not to be
    ```

String Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int length()</td>
<td>Returns the length of this string</td>
</tr>
<tr>
<td>int indexOf(String s)</td>
<td>Returns the index within the string of the first occurrence of the string s</td>
</tr>
<tr>
<td>String substring (int beginx, int endx)</td>
<td>Returns the substring beginning at index beginx and ending at index endx-1</td>
</tr>
<tr>
<td>String toUpperCase()</td>
<td>Converts all characters of the string to uppercase</td>
</tr>
<tr>
<td>String concat(String s)</td>
<td>Concatenates the string s to the end of the original string</td>
</tr>
<tr>
<td>char charAt(int index)</td>
<td>Returns the character at the index, which must be between 0 and length of string - 1</td>
</tr>
</tbody>
</table>

Syntax: primitives vs classes

- Operators act on primitive variables
  - Examples: `+`, `-`, `*`, `%`
  - Standard math in-fix notation
    - `x + y;`
    - `y / 7;`
- Methods act on class variables
  - Example: `length()`
  - Notation: `class.method(arguments)`
    - String `s1` = "foo";
    - `int x = s1.length();`

String Method Examples

```java
import java.util.Scanner;

public class SimpleString {
    public static void main(String[] args) {
        // Keyboard input example
        Scanner keyboard = new Scanner(System.in);
        String string1 = keyboard.nextLine();
        System.out.println(string1);

        // Using String methods
        String string2 = "Hello, world!";
        String reverse = string2.substring(0, 1).concat(string2.substring(2)).concat(string2.substring(1));
        System.out.println(reverse); // prints "Hello, world!
    }
}
```
**Object Examples**

- **Scanner** instance is an object (not primitive)
- Methods for the **Scanner** class include
  - `nextInt` ← returns next sequence as integer
  - `nextDouble` ← returns next sequence as double
  - `next` ← returns next sequence of chars
    - read until the next whitespace (spaces, tabs, end of line)
  - `nextLine` ← returns next line up until enter key
  - reads

**Input/Output (I/O)**

Q: Without a graphical user interface (GUI), how does a program communicate with user?

A: Must use the console for output, keyboard for input.

- Console Output
- Keyboard Input

**Console**

- We’ve seen several examples of console output already.
- `System.out` is the output object in Java.
- Question: Is there a `System.in` for input?
- `println()` is one of the methods available to the `System.out` object.
- `print()` is another method, which is identical to `println()` without line termination.

**Screen Output**

- The concatenation operator (+) is useful when everything does not fit on one line.

```java
System.out.println("Lucky number = "+13+
"Secret number = "+number);
```

- Beware of dual use of + for addition and concatenation:

```java
System.out.println("5+3= "+5+3);
```
Keyboard Input

- Java has reasonable facilities for handling keyboard input.
- These facilities are provided by the Scanner class in the java.util package.
  - A package is a library of classes.

Using the Scanner Class

- Near the beginning of your program, insert
  ```java
  import java.util.Scanner;
  ```
- Create an object of the Scanner class
  ```java
  Scanner keyboard = new Scanner(System.in);
  ```
- Read data (an int or a double, for example)
  ```java
  int n1 = keyboard.nextInt();
  double d1 = keyboard.nextDouble();
  ```

More Scanner Class

- From util package
  ```java
  import java.util.Scanner;
  ```
- Create a new instance:
  ```java
  Scanner in = new Scanner(System.in);
  ```
- Input (depends on data type reading in)
  ```java
  String input = in.next();
  String line = in.nextLine();
  int intVal = in.nextInt();
  double dblVal = in.nextDouble();
  ```

Some Scanner Class Methods

- `Scanner_Object_Name.next()`
  Returns the String value consisting of the next keyboard characters up to, but not including, the first delimiter character. The default delimiters are whitespace characters.
- `Scanner_Object_Name.nextLine()`
  Reads the rest of the current keyboard input line and returns the characters read as a value of type String. Note that the line terminator ‘\n’ is read and discarded; it is not included in the string returned.
- `Scanner_Object_Name.nextInt()`
  Returns the next keyboard input as a value of type int.
- `Scanner_Object_Name.nextDouble()`
  Returns the next keyboard input as a value of type double.
- `Scanner_Object_Name.nextFloat()`
  Returns the next keyboard input as a value of type float.
Reading Integers

```java
import java.util.*;

public class getInput {
    public static void main( String[] args ) {
        Scanner in;
        int intVal;

        in = new Scanner( System.in );
        System.out.println("Enter an integer: ");
        intVal = in.nextInt();

        System.out.println(intVal );
    }
}
```

Reading Strings

```java
import java.util.*;

public class getStringInput {
    public static void main( String[] args ) {
        Scanner in;
        String name;

        in = new Scanner( System.in );
        System.out.println("Enter your name: ");
        name = in.next();

        System.out.println(name );
    }
}
```

nextLine() Method Caution

- The `nextLine()` method reads
  - The remainder of the current line,
  - Even if it is empty.

Example – given following declaration.

```java
int n;
String s1, s2;
int n = scan();
s1 = scan.nextLine();
s2 = scan.nextLine();
```

Assume input shown

- n is set to 42
- but s1 is set to the empty string
- and s2 is set to “next line”
Printing Integers

```java
public class Forecast {
    public static void main (String args[]) {
        System.out.print("The temperature will be ");
        System.out.print(-10);
        System.out.print(" degrees…");
        System.out.println(" and that’s cold, folks!");
    }
}
```

What happens if you use println each time?

Formatting Output with printf

- Java has a method named `printf()` that applies a specific format to output.
- The `printf()` method similar to the `print()/println()` methods.
- `System.out.printf` can have any number of arguments!
  - The first argument contains one or more format specifiers for the remaining arguments
  - All the arguments except the first are values to be output to the screen

printf Format Specifier

- The following code
  ```java
double price = 19.8;
System.out.print("$");
System.out.printf("%6.2f", price);
System.out.println(" each");
```

will output the line

```
$ 19.80 each
```

- The format specifier "%6.2f" is interpreted as follows:
  - Display up to 6 right-justified characters (field width is 6)
  - Display exactly 2 digits after the decimal (precision is 2)
  - Display a floating point number (conversion character is f)

Multiple arguments with printf

- The following code contains a `printf` statement having three arguments
  ```java
double price = 19.8;
String name = "apple";
System.out.printf("$%6.2f per %s.", price, name);
System.out.println();
System.out.println("Wow");
```

will output

```
$ 19.80 per apple.
Wow
```

- Note that the first argument is a format string containing two format specifiers (%6.2f and %s)
- These format specifiers match up with the two arguments that follow (price and name)
Line Breaks with `printf`

- Line breaks can be included in a format string using `%n` or `
`
- The code

  ```
  System.out.printf("Hello%nWorld\n");
  System.out.println("Wow");
  ```

will output

```
Hello
World
!Wow
```

---

Format Specifiers for `System.out.printf`

<table>
<thead>
<tr>
<th>CONVERSION CHARACTER</th>
<th>TYPE OF OUTPUT</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>Decimal (ordinary) integer</td>
<td>%d</td>
</tr>
<tr>
<td>f</td>
<td>Fixed-point (everyday notation) floating point</td>
<td>%f</td>
</tr>
<tr>
<td>e</td>
<td>E-notation floating point</td>
<td>%e</td>
</tr>
<tr>
<td>g</td>
<td>General floating point (Java decides whether to use E-notation or not)</td>
<td>%g</td>
</tr>
<tr>
<td>s</td>
<td>String</td>
<td>%s</td>
</tr>
<tr>
<td>c</td>
<td>Character</td>
<td>%c</td>
</tr>
</tbody>
</table>

---

The `printf` Method (Part 1 of 3)

```java
public class PrintDemo {
    public static void main(String[] args) {
        String aString = "abc";
        System.out.println("String output:");
        System.out.printf("START1234567890 %n", aString);
        System.out.printf("START14END %n", aString);
        System.out.printf("START14END %n", aString);
        System.out.println();
    }
}
```

---

The `printf` Method (Part 2 of 3)

```java
public class PrintDemo {
    public static void main(String[] args) {
        String aString = "abc";
        System.out.println("String output:");
        System.out.printf("START1234567890 %n", aString);
        System.out.printf("START14END %n", aString);
        System.out.printf("START14END %n", aString);
        System.out.println();
    }
}
```
The `printf` Method (Part 3 of 3)

**Legacy Code**
- Code that is "old fashioned" but too expensive to replace is called *legacy code*
- Sometimes legacy code is translated into a more modern language
- The Java method `printf` is just like a C language function of the same name
- This was done intentionally to make it easier to translate C code into Java

**What could go wrong?**
- If you mis-type a variable name or a data type...
  - When you try to compile & run it in Eclipse
    1. Eclipse will tell you there was an error
    2. The editor will put a red ‘x’ at the left of the line with the error
- This is an example of a compile-time error

**What else could go wrong?**
- You can specify an illegal operation
  - E.g. try to divide a string by a string
  - Again, a compile-time error with a red ‘x’
- You can forget a ; or a }
  - Same as above
More Errors

- Capitalization errors
  - Java is case sensitive, identifier names must use the same capitalization rules each time
- Logic Errors
  - Program appears to run correctly, but on closer inspection the wrong output is displayed

Debugging Hints

- Let Eclipse help you!
  - Gives suggestions on methods to use
  - Provides warning and error messages as you type... even provides suggestions of how to fix the problem.
- Add debugging statements to check the computation `System.out.println (...) ;`