BASIC COMPUTATION

OOP Lecture 2

Doyel Pal

OBJECTIVES

Describe the Java data types used for simple data.

Write Java statements to declare variables, define named constants.

Write assignment statements, expressions containing variables and constants.

Define strings of characters, perform simple string processing.

Write Java statements that accomplish keyboard input, screen output.

Adhere to stylistic guidelines and conventions

Write meaningful comments.

OUTLINE

Variables and Expressions

The Class `String`

Keyboard and Screen I/O

Documentation and Style

VARIABLES

Variables store data such as numbers and letters.

- Think of them as places to store data.
- They are implemented as memory locations.

The data stored by a variable is called its *value*.

- The value is stored in the memory location.

Its value can be changed.

VARIABLES

```
Class EggBasket
{
public static void main(String[] args)
{
    int numberOfBaskets = 10;
    int eggsPerBasket = 6;
    int totalEggs = numberOfBaskets * eggsPerBasket ;
    System.out.print("If you have "+eggsPerBasket+" eggs per basket and
      "+numberOfBaskets+" baskets, then the total number of eggs is "+ totalEggs+".");
    }
}
```

```
If you have
6 eggs per basket and
10 baskets, then
the total number of eggs is 60
```

VARIABLES AND VALUES

Variables

numberOfBaskets

eggsPerBasket

totalEggs

Assigning values

```
eggsPerBasket = 6;
numberOfBaskets = 10;
totalEggs = numberOfBaskets * eggsPerBasket;
```

NAMING AND DECLARING VARIABLES

Choose names that are helpful such as `count` or `speed`, but not `c` or `s`.

When you *declare* a variable, you provide its name and type.

```
int numberOfBaskets,eggsPerBasket;
```

A variable's *type* determines what kinds of values it can hold (`int`, `double`, `char`, etc.).

A variable must be declared before it is used.

SYNTAX AND EXAMPLES

Syntax

```
type variable_1, variable_2, …;
```

(`variable_1` is a generic variable called a *syntactic variable*)

Examples

```
int styleChoice, numberOfChecks;
double balance, interestRate;
char jointOrIndividual;
```

DATA TYPES

A *class type* is used for a class of objects and has both data and methods.

- `"Java is fun"` is a value of class type `String`

A *primitive type* is used for simple, nondecomposable values such as an individual number or individual character.

- `int`, `double`, and `char` are primitive types.

PRIMITIVE TYPES

Type Name	Kind of Value	Memory Used	Range of Values
`byte`	Integer	1 byte	–128 to 127
`short`	Integer	2 bytes	–32,768 to 32,767
`int`	Integer	4 bytes	–2,147,483,648 to 2,147,483,647
`long`	Integer	8 bytes	–9,223,372,036,8547,75,808 to 9,223,372,036,854,775,807
`float`	Floating-point	4 bytes	$\pm 3.40282347 \times 10^{+38}$ to $\pm 1.40239846 \times 10^{-45}$
`double`	Floating-point	8 bytes	$\pm 1.79769313486231570 \times 10^{+308}$ to $\pm 4.94065645841246544 \times 10^{-324}$
`char`	Single character (Unicode)	2 bytes	All Unicode values from 0 to 65,535
`boolean`		1 bit	True or false

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.

JAVA IDENTIFIERS

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.

Example keywords: `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 {}.

```
public static void main(String[] args)
{ /* declare variables here */
    . . .
}
```

PRIMITIVE TYPES

Four integer types (`byte`, `short`, `int`, and `long`)

- `int` is most common

Two floating-point types (`float` and `double`)

- `double` is more common

One character type (`char`)

One boolean type (`boolean`)

EXAMPLES OF PRIMITIVE VALUES

Integer types

```
0   -1   365   12000
```

Floating-point types

```
0.99   -22.8   3.14159 5.0
```

Character type

```
'a'   'A'   '#'   ' '
```

Boolean type

```
true   false
```

ASSIGNMENT STATEMENTS

An assignment statement is used to assign a value to a variable.

```
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."

ASSIGNMENT STATEMENTS

Syntax

```
variable = expression
```

where **`expression`** can be another variable, a *literal* or *constant* (such as a number), or something more complicated which combines variables and literals using *operators*

(such as + and -)

ASSIGNMENT EXAMPLES

```
amount = 3.99;
firstInitial = 'W';
score = numberOfCards + handicap;
eggsPerBasket = eggsPerBasket - 2;
```

INITIALIZING VARIABLES

A variable that has been declared, but no yet given a value is said to be *uninitialized*.

Uninitialized class variables have the value `null`.

Uninitialized primitive variables may have a default value.

It's good practice not to rely on a default value.

INITIALIZING VARIABLES

To protect against an uninitialized variable (and to keep the compiler happy), assign a value at the time the variable is declared.

Examples:

```
int count = 0;
```

```
char grade = 'A';
```

syntax

```
type variable_1 = expression_1, variable_2 =
expression_2, …;
```

ASSIGNMENT EVALUATION

The expression on the right-hand side of the assignment operator (=) is evaluated first.

The result is used to set the value of the variable on the left-hand side of the assignment operator.

```
score = numberOfCards +  handicap;
```

```
eggsPerBasket = eggsPerBasket - 2;
```

SIMPLE INPUT

Sometimes the data needed for a computation are obtained from the user at run time.

Keyboard input requires

```
import java.util.Scanner
```

at the beginning of the file.

SIMPLE INPUT

Data can be entered from the keyboard using

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

followed, for example, by

```
eggsPerBasket = keyboard.nextInt();
```

which reads one `int` value from the keyboard and assigns it to `eggsPerBasket`.

SIMPLE INPUT

```
Class EggBasket
{
Public static void main(String[] args)
{
        Scanner input = new Scanner(System.in);
        int numberOfBaskets = input.nextInt();
        int eggsPerBasket = 6;
        int totalEggs = numberOfBaskets * eggsPerBasket ;
        System.out.print("If you have "+eggsPerBasket+" eggs   per basket and
    "+numberOfBaskets+" baskets, then the total number of eggs is "+ totalEggs+".");
}
    }
```

SIMPLE SCREEN OUTPUT

```
System.out.println("The count is " + count);
```

Outputs the sting literal `"the count is "`

Followed by the current value of the variable `count`.

CONSTANTS

Literal expressions such as `2`, `3.7`, or `'y'` are called *constants*.

Integer constants can be preceded by a `+` or `-` sign, but cannot contain commas.

Floating-point constants can be written

- With digits after a decimal point or
- Using e *notation*.

NAMED CONSTANTS

Java provides mechanism to …

- Define a variable
- Initialize it
- Fix the value so it cannot be changed

```
public static final Type Variable = Constant;
```

- Example

```
public static final double PI = 3.14159;
```

ASSIGNMENT COMPATIBILITIES

Java is said to be *strongly typed*.

- You can't, for example, assign a floating point value to a variable declared to store an integer.

Sometimes conversions between numbers are possible.

```
doubleVariable = 7;
```

is possible even if `doubleVariable` is of type `double`, for example.

ASSIGNMENT COMPATIBILITIES

A value of one type can be assigned to a variable of any type further to the right

`byte --> short --> int --> long --> float --> double`

- But not to a variable of any type further to the left.

You can assign a value of type char to a variable of type `int`.

Example : float var = 5;

TYPE CASTING

A *type cast* temporarily 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`.

Any nonzero value to the right of the decimal point is *truncated* rather than *rounded*.

ARITHMETIC OPERATORS

Arithmetic expressions can be formed using the `+`, `-`, `*`, and `/` operators together with variables or numbers referred to as *operands*.

- When both operands are of the same type, the result is of that type.
- When one of the operands is a floating-point type and the other is an integer, the result is a floating point type.

ARITHMETIC OPERATIONS

Example

If `hoursWorked` is an `int` to which the value `40` has been assigned, and `payRate` is a `double` to which `8.25` has been assigned `hoursWorked * payRate` is a `double` with a value of `500.0`.

ARITHMETIC OPERATIONS

Expressions with two or more operators can be viewed as a series of steps, each involving only two operands.

- The result of one step produces one of the operands to be used in the next step.

example

```
balance + (balance * rate)
```

ARITHMETIC OPERATIONS

If at least one of the operands is a floating-point type and the rest are integers, the result will be a floating point type.

The result is the rightmost type from the following list that occurs in the expression.

```
byte --> short --> int --> long --> float --> double
```

THE DIVISION OPERATOR

The division operator (`/`) behaves as expected if one of the operands is a floating-point type.

When both operands are integer types, the result is truncated, not rounded.

- Hence, `99/100` has a value of `0`.

THE MOD OPERATOR

The `mod` (`%`) operator is used with operators of integer type to obtain the remainder after integer division.

14 divided by 4 is 3 *with a remainder of 2*.

- Hence, `14 % 4` is equal to `2`.

The mod operator has many uses, including

- determining if an integer is odd or even
- determining if one integer is evenly divisible by another integer.

PARENTHESES AND PRECEDENCE

Parentheses can communicate the order in which arithmetic operations are performed

examples:

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

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

Without parentheses, an expressions is evaluated according to the *rules of 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.

Highest Precedence

First: the unary operators +, -, !, ++, and --

Second: the binary arithmetic operators *, /, and %

Third: the binary arithmetic operators + and –

Lowest Precedence

PRECEDENCE RULES

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

Even when parentheses are not needed, they can be used to make the code clearer.

```
balance + (interestRate * balance)
```

Spaces also make code clearer

```
balance + interestRate*balance
```

but spaces do not dictate precedence.

SAMPLE EXPRESSIONS

Ordinary Math	Java (Preferred Form)	Java (Parenthesized)
$rate^2 + delta$	`rate * rate + delta`	`(rate * rate) + delta`
$2(salary + bonus)$	`2 * (salary + bonus)`	`2 * (salary + bonus)`
$\frac{1}{time + 3mass}$	`1 / (time + 3 * mass)`	`1 / (time + (3 * mass))`
$\frac{a - 7}{t + 9v}$	`(a - 7) / (t + 9 * v)`	`(a - 7) / (t + (9 * v))`

SPECIALIZED ASSIGNMENT OPERATORS

Assignment operators can be combined with arithmetic operators (including `-`, `*`, `/`, and `%`).

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

can be written as

```
amount += 5;
```

yielding the same results.

INCREMENT AND DECREMENT OPERATORS

Used to increase (or decrease) the value of a variable by 1

Easy to use, important to recognize

The increment operator
`count++` or `++count`

The decrement operator
`count--` or `--count`

INCREMENT AND DECREMENT OPERATORS: TWO OPTIONS

Post-Increment

count++

- Uses current value of variable, THEN increments it

Pre-Increment

++count

- Increments variable first, THEN uses new value

INCREMENT AND DECREMENT OPERATORS

equivalent operations

```
count++;
++count;
count = count + 1;

count--;
--count;
count = count - 1;
```

INCREMENT AND DECREMENT OPERATORS IN EXPRESSIONS

after executing

```
int m = 4;
int result = 3 * (++m)
```

result = 15

m =5

after executing

```
int m = 4;
int result = 3 * (m++)
```

result = 12

m = 5

CASE STUDY

1. Write a program to convert temperature from Fahrenheit to Celsius.

   Take input from user.

   Formula required: Temperature = (temperature – 32)*5/9

2. Write a program to swap two numbers using a third temporary variable.

   Take two inputs from user.

   Example:

   - Input: a = 5, b = 10.
   - Output: a = 10, b = 5.

THE CLASS `STRING`

We've used constants of type `String` already.

```
"Enter a whole number from 1 to 99."
```

A value of type `String` is a

- Sequence of characters
- Treated as a single item.

STRING CONSTANTS AND VARIABLES

Declaring

```
String greeting;
greeting = "Hello!";
```

or

```
String greeting = "Hello!";
```

or

```
String greeting = new String("Hello!");
```

Printing

```
System.out.println(greeting);
```

CONCATENATION OF STRINGS

Two strings are *concatenated* using the + operator.

```
String greeting = "Hello";
String sentence;
sentence = greeting + " officer";
System.out.println(sentence);
```

Any number of strings can be concatenated using the + operator.

CONCATENATING STRINGS AND INTEGERS

```
String solution;
solution = "The answer is " + 42;
System.out.println (solution);
```

```
The answer is 42
```

STRING METHODS

An object of the `String` class stores data consisting of a sequence of characters.

Objects have methods as well as data

The `length()` method returns the number of characters in a particular `String` object.

```
String greeting = "Hello";
int n = greeting.length();
```

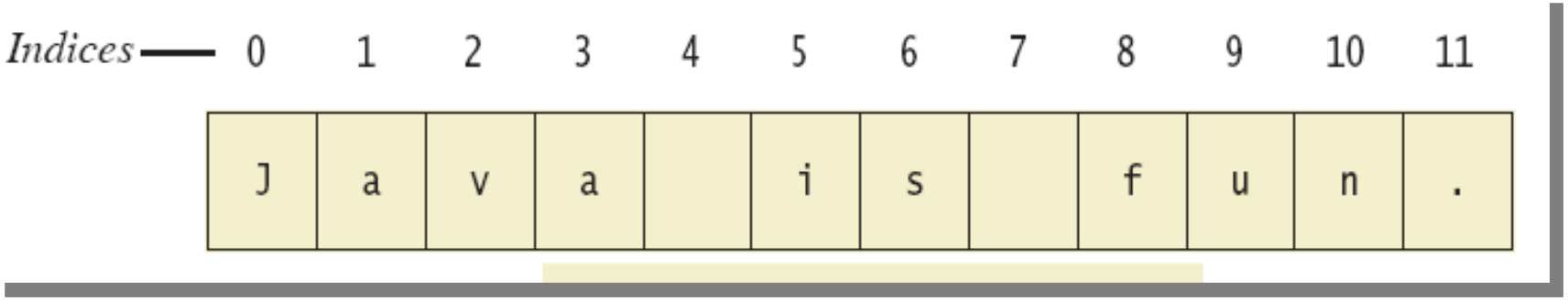
THE METHOD `LENGTH()`

The method `length()` returns an `int`.

You can use a call to method `length()` anywhere an `int` can be used.

```
int count = command.length();
System.out.println("Length is " +command.length());
count = command.length() + 3;
```

`String` Indices



- Positions start with 0, not 1.
  - The 'J' in "Java is fun." is in position 0
- A position is referred to an *index.*
  - The `'f'` in `"Java is fun."` is at index 8.

STRING METHODS

`charAt(`*Index*`)` Returns the character at *Index* in this string. Index numbers begin at 0.
`compareTo(`*A_String*`)` Compares this string with *A_String* to see which string comes first in the lexicographic ordering. (Lexicographic ordering is the same as alphabetical ordering when both strings are either all uppercase letters or all lowercase letters.) Returns a negative integer if this string is first, returns zero if the two strings are equal, and returns a positive integer if *A_String* is first.
`concat(`*A_String*`)` Returns a new string having the same characters as this string concatenated with the characters in *A_String*. You can use the ⇓ operator instead of `concat`.
`equals(`*Other_String*`)` Returns true if this string and *Other_String* are equal. Otherwise, returns false.

STRING METHODS

`equalsIgnoreCase(`*Other_String*`)`

Behaves like the method `equals`, but considers uppercase and lowercase versions of a letter to be the same.

`indexOf(`*A_String*`)`

Returns the index of the first occurrence of the substring *A_String* within this string. Returns -1 if *A_String* is not found. Index numbers begin at 0.

`lastIndexOf(`*A_String*`)`

Returns the index of the last occurrence of the substring *A_String* within this string. Returns -1 if *A_String* is not found. Index numbers begin at 0.

`String` METHODS

`length()` Returns the length of this string.
`toLowerCase()` Returns a new string having the same characters as this string, but with any uppercase letters converted to lowercase.
`toUpperCase()` Returns a new string having the same characters as this string, but with any lowercase letters converted to uppercase.

STRING METHODS

`replace(`*OldChar*, *NewChar*`)` Returns a new string having the same characters as this string, but with each occurrence of *OldChar* replaced by *NewChar*.
`substring(`*Start*`)` Returns a new string having the same characters as the substring that begins at index *Start* of this string through to the end of the string. Index numbers begin at 0.
`substring(`*Start*, *End*`)` Returns a new string having the same characters as the substring that begins at index *Start* of this string through, but not including, index *End* of the string. Index numbers begin at 0.
`trim()` Returns a new string having the same characters as this string, but with leading and trailing whitespace removed.

`String` PROCESSING

No methods allow you to change the value of a `String` object.

But you can change the value of a `String` variable.

```
Text processing is hard!
012345678901234567890123
The word "hard" starts at index 19
The changed string is:
TEXT PROCESSING IS EASY!
```

ESCAPE CHARACTERS

How would you print

```
"Java" refers to a language.
```

The compiler needs to be told that the quotation marks (") do not signal the start or end of a string, but instead are to be printed.

```
System.out.println("\"Java\" refers to a language.");
```

ESCAPE CHARACTERS

`\"`	Double quote.
`\'`	Single quote.
`\\`	Backslash.
`\n`	New line. Go to the beginning of the next line.
`\r`	Carriage return. Go to the beginning of the current line.
`\t`	Tab. Add whitespace up to the next tab stop.

Each escape sequence is a single character even though it is written with two symbols.

EXAMPLES

```
System.out.println("abc\\def");
```

```
abc\def
```

```
System.out.println("new\nline");
```

```
new
line
```

```
char singleQuote = '\'';
System.out.println(singleQuote);
```

```
'
```

KEYBOARD AND SCREEN I/O: OUTLINE

- Screen Output
- Keyboard Input

SCREEN OUTPUT

We've seen several examples of screen output already.

`System.out` is an object that is part of Java.

`println()` is one of the methods available to the `System.out` object.

SCREEN OUTPUT

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

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

Do not break the line except immediately before or after the concatenation operator (+).

SCREEN OUTPUT

Alternatively, use `print()`

```
System.out.print("One, two,");
System.out.print(" buckle my shoe.");
System.out.println(" Three, four,");
System.out.println(" shut the door.");
```

ending with a `println()`.

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

```
import java.util.Scanner;
```

Create an object of the **`Scanner`** class

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

Read data (an **`int`** or a **`double`**, for example)

```
int n1 = keyboard.nextInt();
double d1 = keyboard,nextDouble();
```

KEYBOARD INPUT DEMONSTRATION

```
Enter two whole numbers
separated by one or more spaces:
  42   43
You entered 42 and 43
Next enter two numbers.
A decimal point is OK.
 9.99  21
You entered 9.99 and 21.0
Next enter two words:
plastic spoons
You entered "plastic" and "spoons"
Next enter a line of text:
May the hair on your toes grow long and curly.
You entered "May the hair on your toes grow long and curly."
```

SOME SCANNER CLASS METHODS

Scannner_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.

Scannner_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.

Scannner_Object_Name`.nextInt()`
Returns the next keyboard input as a value of type `int`.

Scannner_Object_Name`.nextDouble()`
Returns the next keyboard input as a value of type `double`.

Scannner_Object_Name`.nextFloat()`
Returns the next keyboard input as a value of type `float`.

SOME SCANNER CLASS METHODS

Scannner_Object_Name`.nextLong()`
Returns the next keyboard input as a value of type `long`.

Scannner_Object_Name`.nextByte()`
Returns the next keyboard input as a value of type `byte`.

Scannner_Object_Name`.nextShort()`
Returns the next keyboard input as a value of type `short`.

Scannner_Object_Name`.nextBoolean()`
Returns the next keyboard input as a value of type `boolean`. The values of `true` and `false` are entered as the words *true* and *false*. Any combination of uppercase and lowercase letters is allowed in spelling *true* and *false*.

Scannner_Object_Name`.useDelimiter(`*Delimiter_Word*`);`
Makes the string *Delimiter_Word* the only delimiter used to separate input. Only the exact word will be a delimiter. In particular, blanks, line breaks, and other whitespace will no longer be delimiters unless they are a part of *Delimiter_Word*.

This is a simple case of the use of the `useDelimiter` method. There are many ways to set the delimiters to various combinations of characters and words, but we will not go into them in this book.

`nextLine()` METHOD CAUTION

The `nextLine()` method reads

- The remainder of the current line,
- Even if it is empty.

NEXTLINE() METHOD CAUTION

Example – given following declaration.

```
int n;
String s1, s2;
n = keyboard.nextInt();
s1 = keyboard.nextLine();
s2 = keyboard.nextLine();
```

Assume input shown

```
42
and don't you
forget it.
```

`n` is set to `42`

but `s1` is set to the empty string.

THE EMPTY STRING

- A string can have any number of characters, including zero.
- The string with zero characters is called the *empty* string.
- The empty string is useful and can be created in many ways including
- `String s3 = "";`

OTHER INPUT DELIMITERS (OPTIONAL)

Almost any combination of characters and strings can be used to separate keyboard input.

to change the delimiter to `"##"`

```
keyboard2.useDelimiter("##");
```

- whitespace will no longer be a delimiter for `keyboard2` input

SOME SCANNER CLASS METHODS

```
Scanner input = new Scanner("MAC 190/ Students /25/Programming");
input.useDelimiter("/");
System.out.println("The delimiter used is "+input.delimiter());

while(input.hasNext()){
        System.out.println(input.next());
}
```

```
Output:
The delimiter used is /
MAC 190
 Students
25
Programming
```

OTHER INPUT DELIMITERS

```
Enter a line of text with two words:
funny wo##rd##
The two words are "funny" and "wor##rd##"
Enter a line of text with two words
delimited by ##:
funny wor##rd##
The two words are "funny wo" and "rd"
```

DOCUMENTATION AND STYLE: OUTLINE

1. Meaningful Names
2. Comments
3. Indentation
4. Named Constants

DOCUMENTATION AND STYLE

- Most programs are modified over time to respond to new requirements.
- Programs which are easy to read and understand are easy to modify.
- Even if it will be used only once, you have to read it in order to debug it .

MEANINGFUL VARIABLE NAMES

A variable's name should suggest its use.

Observe conventions in choosing names for variables.

- Use only letters and digits.
- "Punctuate" using uppercase letters at word boundaries (e.g. `taxRate`).
- Start variables with lowercase letters.
- Start class names with uppercase letters.

COMMENTS

The best programs are self-documenting.

- Clean style
- Well-chosen names

Comments are written into a program as needed explain the program.

- They are useful to the programmer, but they are ignored by the compiler.

COMMENTS

A comment can begin with //.

Everything after these symbols and to the end of the line is treated as a comment and is ignored by the compiler.

```
double radius; //in centimeters
```

COMMENTS

A comment can begin with `/*` and end with `*/`

Everything between these symbols is treated as a comment and is ignored by the compiler.

```
/**
This program should only
be used on alternate Thursdays,
except during leap years, when it should
only be used on alternate Tuesdays.
*/
```

COMMENTS

A *javadoc* comment, begins with `/**` and ends with `*/`.

It can be extracted automatically from Java software.

```
/** method change requires the number of coins to be nonnegative */
```

WHEN TO USE COMMENTS

Begin each program file with an explanatory comment

- What the program does
- The name of the author
- Contact information for the author
- Date of the last modification.

Provide only those comments which the expected reader of the program file will need in order to understand it.

INDENTATION

- Indentation should communicate nesting clearly.
- A good choice is four spaces for each level of indentation.
- Indentation should be consistent.
- Indentation should be used for second and subsequent lines of statements which do not fit on a single line.
- Indentation does not change the behavior of the program.
- Proper indentation helps communicate to the human reader the nested structures of the program.

NAMED CONSTANTS

Once the value of a constant is set (or changed by an editor), it can be used (or reflected) throughout the program.

```
public static final double INTEREST_RATE = 6.65;
```

If a literal (such as 6.65) is used instead, every occurrence must be changed, with the risk than another literal with the same value might be changed unintentionally.

USING NAMED CONSTANTS

To avoid confusion, always name constants (and variables).

```
area = PI * radius * radius;
```

is clearer than

```
area = 3.14159 * radius * radius;
```

Place constants near the beginning of the program.

DECLARING CONSTANTS

Syntax

```
public static final Variable_Type = Constant;
```

Examples

```
public static final double PI = 3.14159;
public static final String MOTTO = "The customer is always right.";
```

By convention, uppercase letters are used for constants.

INPUTTING NUMERIC TYPES

Methods for converting strings to numbers

Result Type	Method for Converting
`byte`	`Byte.parseByte(`*String_To_Convert*`)`
`short`	`Short.parseShort(`*String_To_Convert*`)`
`int`	`Integer.parseInt(`*String_To_Convert*`)`
`long`	`Long.parseLong(`*String_To_Convert*`)`
`float`	`Float.parseFloat(`*String_To_Convert*`)`
`double`	`Double.parseDouble(`*String_To_Convert*`)`

SUMMARY

You have become familiar with Java primitive types (numbers, characters, etc.).

You have learned about assignment statements and expressions.

You have learned about strings.

You have become familiar with classes, methods, and objects.

You have learned about simple keyboard input and screen output.