

# Chapter 1

## Getting Started

Prof. Choonhwa Lee

Dept. of Computer Science and Engineering  
Hanyang University

# Objects and Methods

- Java is an *object-oriented programming (OOP)* language
  - Programming methodology that views a program as consisting of *objects* that interact with one another by means of actions (called *methods*)
  - Objects of the same kind are said to have the same *type* or be in the same *class*

# Terminology Comparisons

- Other high-level languages have constructs called procedures, methods, functions, and/or subprograms
  - These types of constructs are called *methods* in Java
  - All programming constructs in Java, including *methods*, are part of a *class*

# Java Application Programs

- There are two types of Java programs: *applications* and *applets*
- A Java *application program* or "regular" Java program is a class with a method named **main**
  - When a Java application program is run, the *run-time system* automatically invokes the method named **main**
  - All Java application programs start with the **main** method

# Applets

- A Java *applet* (*little Java application*) is a Java program that is meant to be run from a Web browser
  - Can be run from a location on the Internet
  - Can also be run with an applet viewer program for debugging
  - Applets always use a windowing interface
- In contrast, application programs may use a windowing interface or console (i.e., text) I/O

# A Sample Java Application Program

**Display 1.1 A Sample Java Program**

```
1 public class FirstProgram
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Hello reader.");
6         System.out.println("Welcome to Java.");
7
8         System.out.println("Let's demonstrate a simple calculation.");
9         int answer;
10        answer = 2 + 2;
11        System.out.println("2 plus 2 is " + answer);
12    }
```

Annotations:

- ← Name of class (program) (points to `FirstProgram`)
- ← The main method (points to `main`)

## SAMPLE DIALOGUE 1

```
Hello reader.
Welcome to Java.
Let's demonstrate a simple calculation.
2 plus 2 is 4
```

# System.out.println

- Java programs work by having things called *objects* perform actions
  - **System.out**: an object used for sending output to the screen
- The actions performed by an object are called *methods*
  - **println**: the method or action that the **System.out** object performs

# System.out.println

- *Invoking or calling* a method: When an object performs an action using a method
  - Also called *sending a message* to the object
  - Method invocation syntax (in order): an object, a dot (period), the method name, and a pair of parentheses
  - Arguments: Zero or more pieces of information needed by the method that are placed inside the parentheses

```
System.out.println("This is an argument");
```



# Byte-Code and the Java Virtual Machine

- The compilers for most programming languages translate high-level programs directly into the machine language for a particular computer
  - Since different computers have different machine languages, a different compiler is needed for each one
- In contrast, the Java compiler translates Java programs into *byte-code*, a machine language for a fictitious computer called the *Java Virtual Machine*
  - Once compiled to *byte-code*, a Java program can be used on any computer, making it very portable

# Identifiers

- *Identifier*: The name of a variable or other item (class, method, object, etc.) defined in a program
  - A Java identifier must not start with a digit, and all the characters must be letters, digits, or the underscore symbol
  - Java identifiers can theoretically be of any length
  - Java is a case-sensitive language: **Rate**, **rate**, and **RATE** are the names of three different variables

# Identifiers

- Keywords and Reserved words: Identifiers that have a predefined meaning in Java

- Do not use them to name anything else

`public`      `class`      `void`      `static`

- Predefined identifiers: Identifiers that are defined in libraries required by the Java language standard

- Although they can be redefined, this could be confusing and dangerous if doing so would change their standard meaning

`System`      `String`      `println`

# Naming Conventions

- Start the names of variables, methods, and objects with a lowercase letter, indicate "word" boundaries with an uppercase letter, and restrict the remaining characters to digits and lowercase letters

`topSpeed`      `bankRate1`      `timeOfArrival`

- Start the names of classes with an uppercase letter and, otherwise, adhere to the rules above

`FirstProgram`      `MyClass`      `String`

# Variable Declarations

- Every variable in a Java program must be *declared* before it is used
  - A variable declaration tells the compiler what kind of data (type) will be stored in the variable
  - The type of the variable is followed by one or more variable names separated by commas, and terminated with a semicolon
  - Variables are typically declared just before they are used or at the start of a block (indicated by an opening brace { )
  - Basic types in Java are called *primitive types*

```
int numberOfBeans;
```

```
double oneWeight, totalWeight;
```

# Variable Declarations

- Every variable in a Java program must be *declared* before it is used
    - A variable must be declared before it is used
    - The declaration specifies the type of the variable
    - Variables are declared at the start of a method or class
    - Basic types in Java are called *primitive types*
      - Basic types
      - Primitive types**
      - Built-in types
    - Derived types
      - Class types**
      - User-defined types
- ```
int numberOfBeans;  
double oneWeight, totalWeight;
```

# Primitive Types

**Display 1.2    Primitive Types**

| TYPE NAME            | KIND OF VALUE                           | MEMORY USED | SIZE RANGE                                                                               |
|----------------------|-----------------------------------------|-------------|------------------------------------------------------------------------------------------|
| <code>boolean</code> | <code>true</code> or <code>false</code> | 1 byte      | not applicable                                                                           |
| <code>char</code>    | single character (Unicode)              | 2 bytes     | all Unicode characters                                                                   |
| <code>byte</code>    | integer                                 | 1 byte      | −128 to 127                                                                              |
| <code>short</code>   | integer                                 | 2 bytes     | −32768 to 32767                                                                          |
| <code>int</code>     | integer                                 | 4 bytes     | −2147483648 to 2147483647                                                                |
| <code>long</code>    | integer                                 | 8 bytes     | −9223372036854775808 to 9223372036854775807                                              |
| <code>float</code>   | floating-point number                   | 4 bytes     | $-3.40282347 \times 10^{+38}$ to $-1.40239846 \times 10^{-45}$                           |
| <code>double</code>  | floating-point number                   | 8 bytes     | $\pm 1.76769313486231570 \times 10^{+308}$ to $\pm 4.94065645841246544 \times 10^{-324}$ |

# Constants

- *Constant (or literal)*: An item in Java which has one specific value that cannot change
  - Constants of an integer type may not be written with a decimal point (e.g., **10**)
  - Constants of a floating-point type can be written in ordinary decimal fraction form (e.g., **367000.0** or **0.000589**)
  - Constant of a floating-point type can also be written in *scientific (or floating-point) notation* (e.g., **3.67e5** or **5.89e-4**)
    - Note that the number before the **e** may contain a decimal point, but the number after the **e** may not



# Constants

- Constants of type **char** are expressed by placing a single character in single quotes (e.g., '**Z**')
- Constants for strings of characters are enclosed by double quotes (e.g., "**Welcome to Java**")
- There are only two **boolean** type constants, **true** and **false**
  - Note that they must be spelled with all lowercase letters

# The Class **String**

- There is no primitive type for strings in Java
- The class **String** is a predefined class in Java that is used to store and process strings
- Objects of type **String** are made up of strings of characters that are written within double quotes
  - Any quoted string is a constant of type **String**  
**"Live long and prosper."**
- A variable of type **String** can be given the value of a **String** object  
**String blessing = "Live long and prosper.";**

# Concatenation of Strings

- *Concatenation*: Using the `+` operator on two strings in order to connect them to form one longer string
  - If `greeting` is equal to `"Hello "`, and `javaClass` is equal to `"class"`, then `greeting + javaClass` is equal to `"Hello class"`
- Any number of strings can be concatenated together
- When a string is combined with almost any other type of item, the result is a string
  - `"The answer is " + 42` evaluates to `"The answer is 42"`

# Classes, Objects, and Methods

- A *class* is the name for a type whose values are objects
- *Objects* are entities that store data and take actions
  - Objects of the **String** class store data consisting of strings of characters
- The actions that an object can take are called *methods*
  - Methods can return a value of a single type and/or perform an action
  - All objects within a class have the same methods, but each can have different data values

# Classes, Objects, and Methods

- *Invoking or calling a method*: a method is called into action by writing the name of the calling object, followed by a dot, followed by the method name, followed by parentheses
  - This is sometimes referred to as *sending a message to the object*
  - The parentheses contain the information (if any) needed by the method
  - This information is called an *argument* (or *arguments*)

# String Methods

- The **String** class contains many useful methods for string-processing applications
  - A **String** method is called by writing a **String** object, a dot, the name of the method, and a pair of parentheses to enclose any arguments
  - If a **String** method returns a value, then it can be placed anywhere that a value of its type can be used

```
String greeting = "Hello";
int count = greeting.length();
System.out.println("Length is " +
    greeting.length());
```
  - Always count from zero when referring to the *position* or *index* of a character in a string

# String Indexes

## Display 1.5 String Indexes

---

The 12 characters in the string "Java is fun." have indexes 0 through 11.

|   |   |   |   |   |   |   |   |   |   |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| J | a | v | a |   | i | s |   | f | u | n  | .  |

*Notice that the blanks and the period count as characters in the string.*

---

# Some Methods in the Class **String** (Part 1 of 8)

## Display 1.4 Some Methods in the Class String

---

`int` length()

Returns the length of the calling object (which is a string) as a value of type `int`.

### EXAMPLE

After program executes `String greeting = "Hello!";`  
`greeting.length()` returns 6.

`boolean` equals(*Other\_String*)

Returns `true` if the calling object string and the *Other\_String* are equal. Otherwise, returns `false`.

### EXAMPLE

After program executes `String greeting = "Hello";`  
`greeting.equals("Hello")` returns `true`  
`greeting.equals("Good-Bye")` returns `false`  
`greeting.equals("hello")` returns `false`

Note that case matters. "Hello" and "hello" are not equal because one starts with an uppercase letter and the other starts with a lowercase letter.

(continued)



# Some Methods in the Class **String** (Part 2 of 8)

## Display 1.4 Some Methods in the Class String

`boolean equalsIgnoreCase(Other_String)`

Returns true if the calling object string and the *Other\_String* are equal, considering uppercase and lowercase versions of a letter to be the same. Otherwise, returns false.

### EXAMPLE

After program executes `String name = "mary!";`  
`greeting.equalsIgnoreCase("Mary!")` returns `true`

`String toLowerCase()`

Returns a string with the same characters as the calling object string, but with all letter characters converted to lowercase.

### EXAMPLE

After program executes `String greeting = "Hi Mary!";`  
`greeting.toLowerCase()` returns `"hi mary!"`.

(continued)

# Some Methods in the Class **String** (Part 3 of 8)

## Display 1.4 Some Methods in the Class String

---

### String toUpperCase()

Returns a string with the same characters as the calling object string, but with all letter characters converted to uppercase.

#### **EXAMPLE**

After program executes `String greeting = "Hi Mary!";`  
`greeting.toUpperCase()` returns `"HI MARY!"`.

### String trim()

Returns a string with the same characters as the calling object string, but with leading and trailing white space removed. Whitespace characters are the characters that print as white space on paper, such as the blank (space) character, the tab character, and the new-line character `'\n'`.

#### **EXAMPLE**

After program executes `String pause = " Hmm ";`  
`pause.trim()` returns `"Hmm"`.

(continued)

# Some Methods in the Class **String** (Part 4 of 8)

## Display 1.4 Some Methods in the Class String

`char` `charAt(Position)`

Returns the character in the calling object string at the *Position*. Positions are counted 0, 1, 2, etc.

### EXAMPLE

After program executes `String greeting = "Hello!";`  
`greeting.charAt(0)` returns 'H', and  
`greeting.charAt(1)` returns 'e'.

`String` `substring(Start)`

Returns the substring of the calling object string starting from *Start* through to the end of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position *Start* is included in the value returned.

### EXAMPLE

After program executes `String sample = "AbcdefG";`  
`sample.substring(2)` returns "cdefG".

(continued)

# Some Methods in the Class **String** (Part 5 of 8)

## Display 1.4 Some Methods in the Class String

`String substring(Start, End)`

Returns the substring of the calling object string starting from position *Start* through, but not including, position *End* of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position *Start* is included in the value returned, but the character at position *End* is not included.

### EXAMPLE

After program executes `String sample = "AbcdefG";`  
`sample.substring(2, 5)` returns "cde".

`int indexOf(A_String)`

Returns the index (position) of the first occurrence of the string *A\_String* in the calling object string. Positions are counted 0, 1, 2, etc. Returns -1 if *A\_String* is not found.

### EXAMPLE

After program executes `String greeting = "Hi Mary!";`  
`greeting.indexOf("Mary")` returns 3, and  
`greeting.indexOf("Sally")` returns -1.

(continued)

# Some Methods in the Class **String** (Part 6 of 8)

## Display 1.4 Some Methods in the Class String

```
int indexOf(A_String, Start)
```

Returns the index (position) of the first occurrence of the string *A\_String* in the calling object string that occurs at or after position *Start*. Positions are counted 0, 1, 2, etc. Returns -1 if *A\_String* is not found.

### EXAMPLE

After program executes `String name = "Mary, Mary quite contrary";`  
`name.indexOf("Mary", 1)` returns 6.  
The same value is returned if 1 is replaced by any number up to and including 6.  
`name.indexOf("Mary", 0)` returns 0.  
`name.indexOf("Mary", 8)` returns -1.

```
int lastIndexOf(A_String)
```

Returns the index (position) of the last occurrence of the string *A\_String* in the calling object string. Positions are counted 0, 1, 2, etc. Returns -1, if *A\_String* is not found.

### EXAMPLE

After program executes `String name = "Mary, Mary, Mary quite so";`  
`greeting.indexOf("Mary")` returns 0, and  
`name.lastIndexOf("Mary")` returns 12.

(continued)

# Some Methods in the Class **String** (Part 7 of 8)

## Display 1.4 Some Methods in the Class **String**

```
int compareTo(A_String)
```

Compares the calling object string and the string argument to see which comes first in the lexicographic ordering. Lexicographic order is the same as alphabetical order but with the characters ordered as in Appendix 3. Note that in Appendix 3 all the uppercase letters are in regular alphabetical order and all the lowercase letters are in alphabetical order, but all the uppercase letters precede all the lowercase letters. So, lexicographic ordering is the same as alphabetical ordering provided both strings are either all uppercase letters or both strings are all lowercase letters. If the calling string is first, it returns a negative value. If the two strings are equal, it returns zero. If the argument is first, it returns a positive number.

### EXAMPLE

After program executes `String entry = "adventure";`  
`entry.compareTo("zoo")` returns a negative number,  
`entry.compareTo("adventure")` returns 0, and  
`entry.compareTo("above")` returns a positive number.

(continued)

# Some Methods in the Class **String** (Part 8 of 8)

## Display 1.4 Some Methods in the Class String

```
int compareToIgnoreCase(A_String)
```

Compares the calling object string and the string argument to see which comes first in the lexicographic ordering, treating uppercase and lowercase letters as being the same. (To be precise, all uppercase letters are treated as if they were their lowercase versions in doing the comparison.) Thus, if both strings consist entirely of letters, the comparison is for ordinary alphabetical order. If the calling string is first, it returns a negative value. If the two strings are equal ignoring case, it returns zero. If the argument is first, it returns a positive number.

### **EXAMPLE**

After program executes `String entry = "adventure";`  
`entry.compareToIgnoreCase("Zoo")` returns a negative number,  
`entry.compareToIgnoreCase("Adventure")` returns 0, and  
`"Zoo".compareToIgnoreCase(entry)` returns a positive number.



# Display 1.7 Using the **String** Class

```
public class StringProcessingDemo
{
    public static void main(String[] args)
    {
        String sentence = "I hate text processing!";
        int position = sentence.indexOf("hate");
        String ending = sentence.substring(position + "hate".length( ));

        System.out.println("01234567890123456789012");
        System.out.println(sentence);
        System.out.println("The word \"hate\" starts at index " + position);

        sentence = sentence.substring(0, position) + "adore" + ending;
        System.out.println("The changed string is:");
        System.out.println(sentence);
    }
}
```

```
01234567890123456789012
I hate text processing!
The word "hate" starts at index 2
The changed string is:
I adore text processing!
```



# Comments

- A *line comment* begins with the symbols `//`, and causes the compiler to ignore the remainder of the line
  - This type of comment is used for the code writer or for a programmer who modifies the code
- A *block comment* begins with the symbol pair `/*`, and ends with the symbol pair `*/`
  - The compiler ignores anything in between
  - This type of comment can span several lines
  - This type of comment provides documentation for the users of the program

# Comments and a Named Constant

## Display 1.8 Comments and a Named Constant

```
1  /**
2   Program to show interest on a sample account balance.
3   Author: Jane Q. Programmer.
4   E-mail Address: janeq@somemachine.etc.etc.
5   Last Changed: September 21, 2004.
6  */
7  public class ShowInterest
8  {
9      public static final double INTEREST_RATE = 2.5;
10
11     public static void main(String[] args)
12     {
13         double balance = 100;
14         double interest; //as a percent
15
16         interest = balance * (INTEREST_RATE/100.0);
17         System.out.println("On a balance of $" + balance);
18         System.out.println("you will earn interest of $"
19                             + interest);
20         System.out.println("All in just one short year.");
21     }
22 }
```

*Although it would not be as clear, it is legal to place the definition of INTEREST\_RATE here instead.*

### SAMPLE DIALOGUE

On a balance of \$100.0  
you will earn interest of \$2.5  
All in just one short year.