# Chapter 2 - Introduction to Java Applications

## 2.1 Introduction

- In this chapter
  - Introduce examples to illustrate features of Java
  - Two program styles - applications and applets

## 2.2 A Simple Program: Printing a Line of Text

- Application
  - Program that executes using the **`java`** interpreter
- Sample program
  - Show program, then analyze each line

```java
// Fig. 2.1: Welcome1.java
// A first program in Java.

public class Welcome1 {

// main begins execution of Java application
  public static void main( String args[] )
  {
     System.out.println("Welcome to Java
 Program");

 }  // end method main

}  // end class Welcome1
```

**Welcome1.java**

```
Welcome to Java Program
```

**Program Output**

## 2.2 A Simple Program: Printing a Line of Text

```java
   // Fig. 2.1: Welcome1.java
```

- Comments start with: `//`
  - Comments ignored during program execution
  - Document and describe code
  - Provides code readability
- Multiple line comments: `/* ... */`

```
/* This is a multiple
   line comment. It can
   be split over many lines */
```

## 2.2 A Simple Program: Printing a Line of Text

```
3
```

- – Blank line
  - Makes program more readable
  - Blank lines, spaces, and tabs are white-space characters
    - – Ignored by compiler

```
4    public class Welcome1 {
```

- – Begins class definition for class **`Welcome1`**
  - Every Java program has at least one user-defined class
  - One Java program just can have one public class!!!
  - Keyword: words reserved for use by Java
    - – **`class`** keyword followed by class name
  - Naming classes: capitalize every word
    - – **`SampleClassName`**

5

## 2.2 A Simple Program: Printing a Line of Text

```
4    public class Welcome1 {
```

- – Name of class called identifier
  - Series of characters consisting of letters, digits, underscores ( _ ) and dollar signs ( **$**,**￥**, **£** ), and Unicode (Chinese) character!!!
  - Does not begin with a digit, has no spaces
  - Examples: **`Welcome1`**, **`$value`**, **`_value`**, **`button7 is valid`**
    - – **`7button`** is invalid
  - Java is case sensitive (capitalization matters)
    - – **`a1`** and **`A1`** are different

6

## 2.2 A Simple Program: Printing a Line of Text

```java
public class Welcome1 {
```

- Saving files
  - File name must be class name with **`.java`** extension
  - **`Welcome1.java`**

```java
public static void main( String args[]) {
```

- Part of every Java application
  - Applications begin executing at **`main`**
    - Parenthesis indicate **`main`** is a method
- Java applications contain one or more methods
  - Exactly one method must be called **`main`**
- Methods can perform tasks and return information
  - **`void`** means **`main`** returns no information
  - For now, mimic **`main`**'s first line

## 2.2 A Simple Program: Printing a Line of Text

```java
System.out.println("Welcome to Java Program");
```

- Instructs computer to perform an action
  - Prints string of characters
    - String - series characters inside double quotes
  - White-spaces in strings are not ignored by compiler
- **`System.out`**
  - Standard output object
  - Print to command window (i.e., MS-DOS prompt)
- Method **`System.out.println`**
  - Displays line of text
  - Argument inside parenthesis
- This line known as a statement
  - Statements must end with semicolon **`;`**

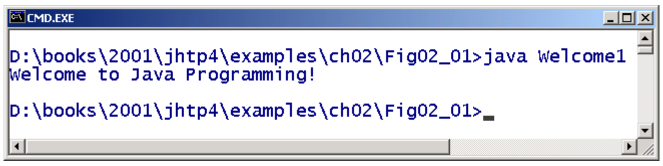
## 2.2.1 Compile and Execute the Java Application

- Compiling a program
  - Open a command prompt window, go to directory where program is stored
  - Type **`javac Welcome1.java`**
  - If no errors, **`Welcome1.class`** created
    - Has bytecodes that represent application
    - Bytecodes passed to Java interpreter

## 2.2.1 Compile and Execute the Java Application

- Executing a program
  - Type **`java Welcome1`**
    - Interpreter loads **`.class`** file for class **`Welcome1`**
    - **`.class`** extension omitted from command
  - Interpreter calls method **`main`**

Fig. 2.2 Executing Welcome1 in a Microsoft Windows 2000 Command Prompt.



```
D:\books\2001\jhtp4\examples\ch02\Fig02_01>java Welcome1
Welcome to Java Programming!

D:\books\2001\jhtp4\examples\ch02\Fig02_01>_
```

## 2.3 Display a Single Text with Multiple Statements

- Modifying programs
  - Welcome2.java (Fig. 2.3) produces same output as Welcome1.java (Fig. 2.1)
  - Using different code

```
9        System.out.println( "Welcome to " );
10       System.out.println( "Java Programming!" );
```

  - Line 9 displays “Welcome to ” with cursor remaining on printed line
  - Line 10 displays “Java Programming! ” on same line with cursor on next line

11

```
1   // Fig. 2.3: Welcome2.java
2   // Printing a line of text with multiple statements.
3
4   public class Welcome2 {
5
6      // main method begins execution of Java application
7      public static void main( String args[] )
8      {
9         System.out.print( "Welcome to " );
10        System.out.println( "Java Programming!" );
11
12     }  // end method main
13
14  }  // end class Welcome2
```

**System.out.print** keeps the cursor on the same line, so **System.out.println** continues on the same line.

```
Welcome to Java Programming!
```

**Welcome2.java**

**1. Comments**

**2. Blank line**

**3. Begin class Welcome2**

**3.1 Method main**

**4. Method System.out.print**

**4.1 Method System.out.println**

**5. end main, Welcome2**

**Program Output**

12

## 2.3.2 Display Multiple Lines with a Single Statement

- Newline characters (\n)
  - Interpreted as “special characters” by methods `System.out.print` and `System.out.println`
  - Indicates cursor should be on next line
  - `Welcome3.java` (Fig. 2.4)

```
      System.out.println( "Welcome\nto\nJava\nProgramming!" );
```

  - Line breaks at \n
- Usage
  - Can use in `System.out.println` or `System.out.print` to create new lines
    - `System.out.println( "Welcome\nto\nJava\nProgramming!" );`

```
// Fig. 2.4: Welcome3.java
// Printing multiple lines of text with a single statement.

public class Welcome1 {

   // main method begins execution of Java application
   public static void main( String args[] )
   {
      System.out.println( "Welcome\nto\nJava\nProgramming!" );

   }  // end method main

}  // end class Welcome3
```

**`Welcome3.java`**

**1. `main`**

**2. `System.out.println` (uses `\n` for new line)**

```
Welcome
to
Java
Programming!
```

**Program Output**

Notice how a new line is output for each **`\n`** escape sequence.

## 2.3.2 Display Multiple Lines with a Single Statement

Escape characters

- Backslash ( \ ) ➔ \r, \f, \n, \b, \t, \\, \’, \”
- Indicates special characters be output

| Escape sequence | Description |
|---|---|
| `\n` | Newline. Position the screen cursor to the beginning of the next line. |
| `\t` | Horizontal tab. Move the screen cursor to the next tab stop. |
| `\r` | Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line. Any characters output after the carriage return overwrite the characters previously output on that line. |
| `\\` | Backslash. Used to print a backslash character. |
| `\"` | Double quote. Used to print a double-quote character. For example, `System.out.println( "\"in quotes\"" );` displays `"in quotes"` |

Fig. 2.5 Some common escape sequences.

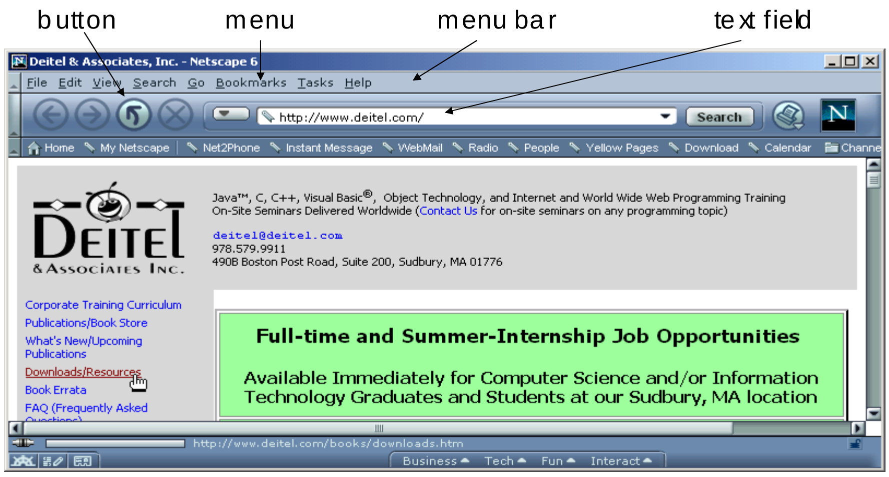
15

## 2.4 Display Text in a Dialog Box

- Display
  - Most Java applications use windows or a dialog box
    - We have used command window
  - Class **`JOptionPane`** allows us to use dialog boxes ➔ swing
- Packages
  - Set of predefined classes for us to use
  - Groups of related classes called *packages*
    - Group of all packages known as Java class library or Java applications programming interface (Java API)
  - **`JOptionPane`** is in the **`javax.swing`** package
    - Package has classes for using Graphical User Interfaces (GUIs)

16

## 2.4 Displaying Text in a Dialog Box


button
menu
menu bar
text field
Deitel & Associates, Inc. - Netscape 6
File Edit View Search Go Bookmarks Tasks Help
http://www.deitel.com/
Search
Home My Netscape Net2Phone Instant Message WebMail Radio People Yellow Pages Download Calendar Channe
Business Tech Fun Interact

DEITEL & ASSOCIATES INC.

Java™, C, C++, Visual Basic®, Object Technology, and Internet and World Wide Web Programming Training
On-Site Seminars Delivered Worldwide (Contact Us for on-site seminars on any programming topic)

deitel@deitel.com
978.579.9911
490B Boston Post Road, Suite 200, Sudbury, MA 01776

Corporate Training Curriculum
Publications/Book Store
What's New/Upcoming Publications
Downloads/Resources
Book Errata
FAQ (Frequently Asked Questions)

**Full-time and Summer-Internship Job Opportunities**

Available Immediately for Computer Science and/or Information Technology Graduates and Students at our Sudbury, MA location

http://www.deitel.com/books/downloads.htm

```java
// Fig. 2.6: Welcome4.java
// Printing multiple lines in a dialog box

// Java extension packages
import javax.swing.JOptionPane;  // import class JOptionPane

public class Welcome4 {

   // main method begins execution of Java application
   public static void main( String args[] )
   {
      JOptionPane.showMessageDialog(
         null, "Welcome\nto\nJava\nProgramming!" );

      System.exit( 0 );  // terminate application

   }  // end method main

}  // end class Welcome4
```

`Welcome4.java`

1. `import` statement

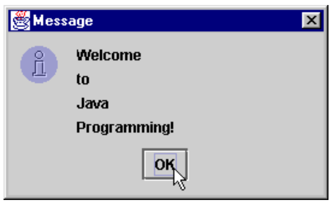
2. Class `Welcome4`

2.1 `main`

2.2 `showMessageDialog`

2.3 `System.exit`

Program Output



```
Welcome
to
Java
Programming!
```

## 2.4 Display Text in a Dialog Box

```java
// Java extension packages
```

- Two groups of packages in Java API
- Core packages
  - Begin with **`java`**
  - Included with Java 2 Software Development Kit
- Extension packages
  - Begin with **`javax`**
  - New Java packages

```java
import javax.swing.JOptionPane;
```

- **`import`** statements
  - Used by compiler to identify and locate classes used in Java programs
  - Tells compiler to load class **`JOptionPane`** from **`javax.swing`** package

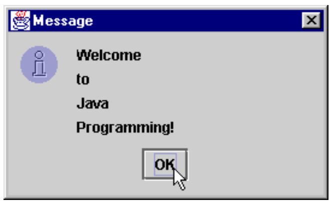
## 2.4 Display Text in a Dialog Box

```java
        JOptionPane.showMessageDialog(
            null, "Welcome\nto\nJava\nProgramming!" );
```

- Call method **`showMessageDialog`** of class **`JOptionPane`**
  - Requires two arguments
  - Multiple arguments separated by commas (**`,`**)
  - For now, first argument always **`null`**
  - Second argument is string to display
- **`showMessageDialog`** is a **`static`** method of class **`JOptionPane`**
  - **`static`** methods called using class name, dot (**`.`**) then method name

## 2.4 Display Text in a Dialog Box

- All statements end with **`;`**
  - A single statement can span multiple lines
  - Cannot split statement in middle of identifier or string
- Executing lines 12 and 13 displays the dialog box



  - Automatically includes an **OK** button
    - Hides or dismisses dialog box
  - Title bar has string **Message**

21

## 2.4 Display Text in a Dialog Box

```
15        System.exit( 0 );
```

- Calls **`static`** method **`exit`** of class **`System`**
  - Terminates application
    - Use with any application displaying a GUI
  - Because method is **`static`**, needs class name and dot (**`.`**)
  - Identifiers starting with capital letters usually class names
- Argument of **`0`** means application ended successfully
  - Non-zero usually means an error occurred
- Class **`System`** part of package **`java.lang`**
  - No **`import`** statement needed
  - **`java.lang`** automatically imported in every Java program

22

## Java keyword and reserved word

- Basic: boolean, byte, char, short, int, long, float, double, void
- Flow-control: if, else, do, while, for, switch, case, default, break, continue
- Access: import, package, public, protected, private
- Modifier: abstract, final, native, volatile, static, strictfp, synchronized, transient
- Inheritance: class, extends, implements, interface, super, this
- Exception control: assert, throw, throws, try, catch, finally
- Other: instanceof, new, return

23

```
// Fig. 2.9: Addition.java
// An addition program.

// Java extension packages
import javax.swing.JOptionPane;  // import class JOptionPane

public class Addition {

   // main method begins exec
   public static void main( S
   {
      String firstNumber;   // first string entered by user
      String secondNumber;  // second string entered by user
      int number1;          // first number to add
      int number2;          // second number to add
      int sum;              // sum of number1 and number2

      // read in first number from user as a String
      firstNumber =
         JOptionPane.showInputDialog( "Enter first integer" );

      // read in second number from user as a String
      secondNumber =
         JOptionPane.showInputDialog( "E

      // convert numbers from type Str
      number1 = Integer.parseInt( firstNumber );
      number2 = Integer.parseInt( secondNumber );

      // add the numbers
      sum = number1 + number2;

```

Input first integer as a **`String`**, assign to **`firstNumber`**.

Convert strings to integers.

Add, place result in **`sum`**.

**`Addition.java`**

1. **`import`**

2. class **`Addition`**

2.1 Declare variables (name and data type)

3. **`InputDialog`**

4. **`parseInt`**

5. Add numbers, put result in **`sum`**

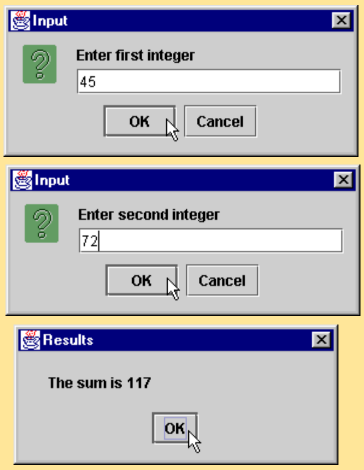
24

```
      // display the results
      JOptionPane.showMessageDialog(
         null, "The sum is " + sum, "Results",
         JOptionPane.PLAIN_MESSAGE );

      System.exit( 0 );   // terminate application

   }  // end method main

}  // end class Addition
```



The sum is 117

**Program output**

## 2.5 Another Java Application: Adding Integers

```
import javax.swing.JOptionPane;
```

- Location of **`JOptionPane`** for use in the program

```
      String firstNumber;   // first string entered by user
      String secondNumber;  // second string entered by user
String firstNumber, secondNumber;
```

- Declaration
  - **`firstNumber`** and **`secondNumber`** are variables
- Variables
  - Location in memory that stores a value
    - Declare with name and data type before use
  - **`firstNumber`** and **`secondNumber`** are of data type **`String`** (package **`java.lang`**)
    - Hold strings
  - Variable name: any valid identifier and Declarations end with semicolons **`;`**

## 2.5 Another Java Application: Adding Integers

```
      int number1;          // first number to add
      int number2;          // second number to add
      int sum;              // sum of number1 and number2
```

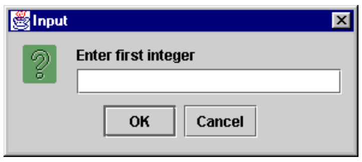
– Declares variables **`number1`**, **`number2`**, and **`sum`** of type **`int`**
  - **`int`** holds integer values (whole numbers): i.e., **`0`**, **`-4`**, **`97`**
  - Data types **`float`** and **`double`** can hold decimal numbers
  - Data type **`char`** can hold a single character: i.e., x, $, \n, 7

27

## 2.5 Another Java Application: Adding Integers

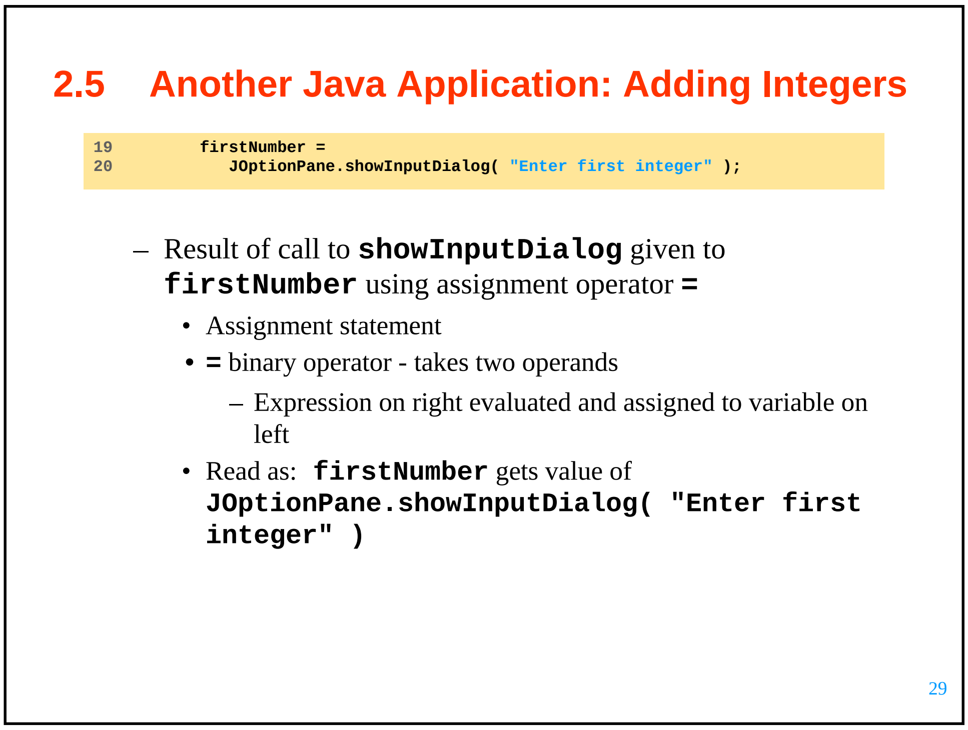
```
      firstNumber =
         JOptionPane.showInputDialog( "Enter first integer" );
```

– Reads **`String`** from the user, representing the first number to be added
  - Method **`JOptionPane.showInputDialog`** displays the following:



  - Message called a prompt - directs user to perform an action
  - Argument appears as prompt text
  - If wrong type of data entered (non-integer) or click **Cancel**, error occurs

28

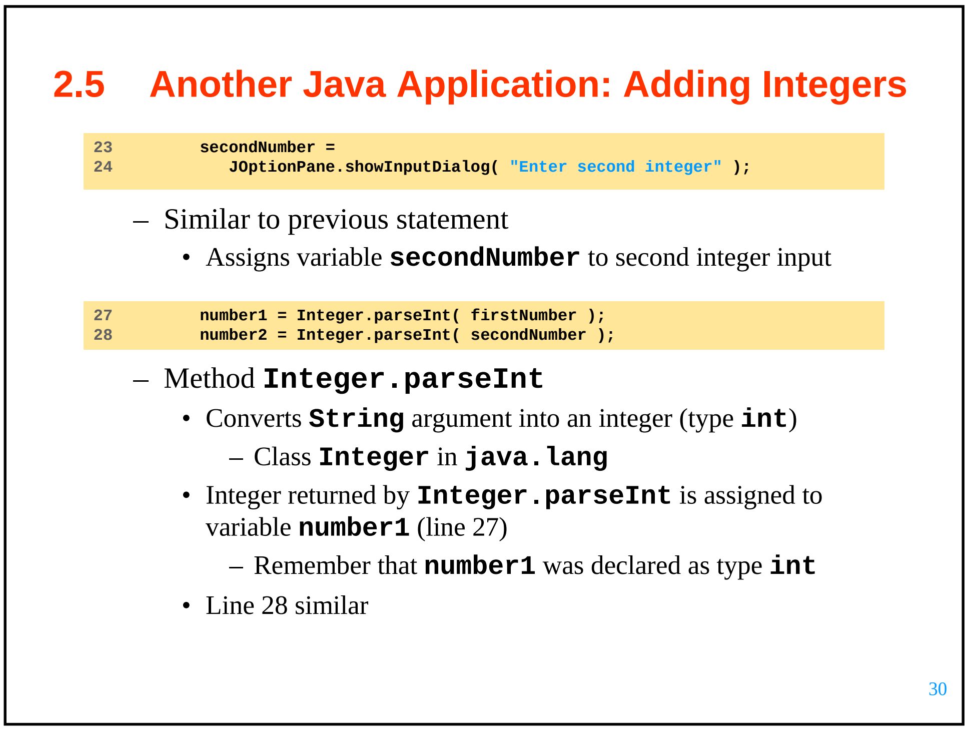


## 2.5 Another Java Application: Adding Integers

```
19         firstNumber =
20            JOptionPane.showInputDialog( "Enter first integer" );
```

- Result of call to **`showInputDialog`** given to **`firstNumber`** using assignment operator **`=`**
  - Assignment statement
  - **`=`** binary operator - takes two operands
    - Expression on right evaluated and assigned to variable on left
  - Read as: **`firstNumber`** gets value of **`JOptionPane.showInputDialog( "Enter first integer" )`**

29



## 2.5 Another Java Application: Adding Integers

```
23         secondNumber =
24            JOptionPane.showInputDialog( "Enter second integer" );
```

- Similar to previous statement
  - Assigns variable **`secondNumber`** to second integer input

```
27         number1 = Integer.parseInt( firstNumber );
28         number2 = Integer.parseInt( secondNumber );
```

- Method **`Integer.parseInt`**
  - Converts **`String`** argument into an integer (type **`int`**)
    - Class **`Integer`** in **`java.lang`**
  - Integer returned by **`Integer.parseInt`** is assigned to variable **`number1`** (line 27)
    - Remember that **`number1`** was declared as type **`int`**
  - Line 28 similar

30

## 2.5 Another Java Application: Adding Integers

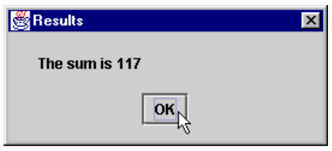
```
34        JOptionPane.showMessageDialog(
35           null, "The sum is " + sum, "Results",
36           JOptionPane.PLAIN_MESSAGE );
```

- Use **`showMessageDialog`** to display results
- **`"The sum is " + sum`**
  - Uses the operator + to "add" the string literal **`"The sum is"`** and **`sum`**
  - Concatenation of a **`String`** and another data type
    - Results in a new string
  - If **`sum`** contains **`117`**, then **`"The sum is " + sum`** results in the new string **`"The sum is 117"`**
  - Note the space in **`"The sum is "`**

## 2.5 Another Java Application: Adding Integers

```
34        JOptionPane.showMessageDialog(
35           null, "The sum is " + sum, "Results",
36           JOptionPane.PLAIN_MESSAGE );
```

- Different version of **`showMessageDialog`**
  - Requires four arguments (instead of two as before)
  - First argument: **`null`** for now
  - Second: string to display
  - Third: string in title bar
  - Fourth: type of message dialog with icon
    - Line 36 no icon: **`JoptionPane.PLAIN_MESSAGE`**



The sum is 117

## 2.5 Another Java Application: Adding Integers

| Message dialog type | Icon | Description |
|---|---|---|
| `JOptionPane.ERROR_MESSAGE` | 🛑 | Displays a dialog that indicates an error to the user. |
| `JOptionPane.INFORMATION_MESSAGE` | ℹ | Displays a dialog with an informational message to the user. The user can simply dismiss the dialog. |
| `JOptionPane.WARNING_MESSAGE` | ⚠ | Displays a dialog that warns the user of a potential problem. |
| `JOptionPane.QUESTION_MESSAGE` | ? | Displays a dialog that poses a question to the user. This dialog normally requires a response, such as clicking on a **Yes** or a **No** button. |
| `JOptionPane.PLAIN_MESSAGE` | no icon | Displays a dialog that simply contains a message, with no icon. |

Fig. 2.12 **`JOptionPane`** constants for message dialogs.

## Java operator

- C like operator
  - +, -, *, /, %, ++, --, *=…
  - >=, <=, ==, !=, ? Stmt1 : stmt2
  - &&, ||, !
  - &, |, ~, ^, <<, >>, >>> (unsigned right shift)
- C like data type
  - Byte (-128 – 127)
  - Short (-32768 – 32767)
  - Char
  - Int (-2147483648 – 2147483647)
  - Long ($-2^{63}$ – $2^{63}$-1)
  - float
  - double

## Java confusing!!!

- int a=4;  int a=06;  int a=0x1a;
- Long a=4L; long a=6L;
- Float a=1.4f; float a=8.0F; float a=4.0; (X)
- Char c1=‘a’; char c2=‘\n’; char c3=26131; char c4=‘\u74cf’; (Unicode)
- Conversion !!
  - byte ➔ short ➔ int ➔ long ➔ float ➔ double
  - char ➔ int ➔ long ➔ float ➔ double

35

## 2.7 Arithmetic

- Arithmetic calculations used in most programs
  - Usage
    - `*` for multiplication
    - `/` for division
    - `+`, `-`
    - No operator for exponentiation (more in Chapter 5)
  - Integer division truncates remainder

    `7 / 5` evaluates to `1`
  - Modulus operator `%` returns the remainder

    `7 % 5` evaluates to `2`

36

## 2.7 Arithmetic

| Operator(s) | Operation(s) | Order of evaluation (precedence) |
|---|---|---|
| `( )` | Parentheses | Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses on the same level (i.e., not nested), they are evaluated left to right. |
| `*`, `/` and `%` | Multiplication<br>Division<br>Modulus | Evaluated second. If there are several of this type of operator, they are evaluated from left to right. |
| `+` or `-` | Addition<br>Subtraction | Evaluated last. If there are several of this type of operator, they are evaluated from left to right. |

**Fig. 2.17** Precedence of arithmetic operators.

## 2.8 Decision Making: Equality and Relational Operators

- **`if`** control structure
  - Simple version in this section, more detail later
  - If a condition is true, then the body of the **`if`** statement executed
    - **`0`** interpreted as false, non-zero is true (X just C used)
    - Java use boolean value (true, false) for if condition
  - Control always resumes after the **`if`** structure
  - Conditions for **`if`** structures can be formed using equality or relational operators (next slide)

```
if ( condition )
   statement executed if condition true
```

  - No semicolon needed after condition
    - Else conditional task not performed

## 2.8 Decision Making: Equality and Relational Operators

| Standard algebraic equality or relational operator | Java equality or relational operator | Example of Java condition | Meaning of Java condition |
|---|---|---|---|
| *Equality operators* | | | |
| = | **==** | **x == y** | **x** is equal to **y** |
| | **!=** | **x != y** | **x** is not equal to **y** |
| *Relational operators* | | | |
| > | **>** | **x > y** | **x** is greater than **y** |
| < | **<** | **x < y** | **x** is less than **y** |
| | **>=** | **x >= y** | **x** is greater than or equal to **y** |
| ? | **<=** | **x <= y** | **x** is less than or equal to **y** |

**Fig. 2.19** Equality and relational operators.

- Upcoming program uses **`if`** structures
  - Discussion afterwards

```java
// Fig. 2.20: Comparison.java
// Compare integers using if structures, relational operators
// and equality operators.

// Java extension packages
import javax.swing.JOptionPane;

public class Comparison {

   // main method begins execution of Java application
   public static void main( String args[] )
   {
      String firstNumber;   // first string entered by user
      String secondNumber;  // second string entered by user
      String result;        // a string containing the output
      int number1;          // first number to compare
      int number2;          // second number to compare

      // read first number from user as a string
      firstNumber =
         JOptionPane.showInputDialog( "Enter first integer:" );

      // read second number from user as a string
      secondNumber =
         JOptionPane.showInputDialog( "Enter second integer:" );

      // convert numbers from type String to type int
      number1 = Integer.parseInt( firstNumber );
      number2 = Integer.parseInt( secondNumber );

      // initialize result to empty String
      result = "";

```

**`Comparison.java`**

1. `import`

2. Class `Comparison`

2.1 `main`

2.2 Declarations

2.3 Input data (`showInputDialog`)

2.4 `parseInt`

2.5 Initialize `result`

```java
      if ( number1 == number2 )
         result = number1 + " == " + number2;

      if ( number1 != number2 )
         result = number1 + " != " + number2;

      if ( number1 < number2 )
         result = result + "\n" + number1 + " < " + number2;

      if ( number1 > number2 )
         result = result + "\n" + number1 + " > " + number2;

      if ( number1 <= number2 )
         result = result + "\n" + number1 + " <= " + number2;

      if ( number1 >= number2 )
         result = result + "\n" + number1 + " >= " + number2;

      // Display results
      JOptionPane.showMessageDialog(
         null, result, "Comparison Results",
         JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );  // terminate application

   }  // end method main

}  // end class Comparison
```

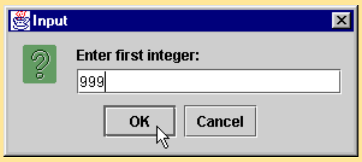
Test for equality, create new string, assign to `result`.

Notice use of `JOptionPane.INFORMATION_MESSAGE`

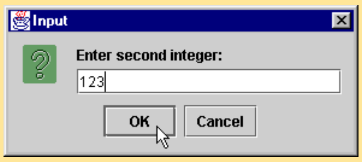
`Comparison.java`

3. `if` statements

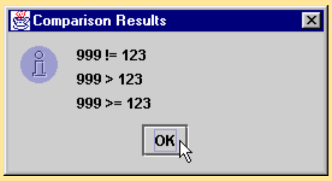
4. `showMessageDialog`



```
Input
Enter first integer:
999
OK   Cancel
```



```
Input
Enter second integer:
123
OK   Cancel
```



```
Comparison Results
999 != 123
999 > 123
999 >= 123
OK
```

**Program Output**

## 2.8 Decision Making: Equality and Relational Operators

```
32        result = "";
```

– Initialize **`result`** with empty string

```
34        if ( number1 == number2 )
35           result = number1 + " == " + number2;
```

– **`if`** structure to test for equality using (**`==`**)
  - If variables equal (condition true)
    - **`result`** concatenated using **`+`** operator
    - **`result = result + other strings`**
    - Right side evaluated first, new string assigned to **`result`**
  - If variables not equal, statement skipped

## 2.8 Decision Making: Equality and Relational Operators

– Lines 37-50: other **`if`** structures testing for less than, more than, etc.
  - If number1 = 123 and number2 = 123
    - Line 34 evaluates true (if number1 = = number 2)
      - Because number1 equals number2
    - Line 40 evaluates false (if number1 < number 2)
      - Because number1 is not less than number2
    - Line 49 evaluates true (if number1 >= number2)
      - Because number1 is greater than or equal to number2

## 2.8 Decision Making: Equality and Relational Operators

- Precedence of operators
  - All operators except for **=** (assignment) associates from left to right
    - For example: **x = y = z** is evaluated **x = (y = z)**

| Operators | Associativity | Type |
|---|---|---|
| **()** | left to right | parentheses |
| *** / %** | left to right | multiplicative |
| **+ -** | left to right | additive |
| **< <= > >=** | left to right | relational |
| **== !=** | left to right | equality |
| **=** | right to left | assignment |

**Fig. 2.21** Precedence and associativity of the operators discussed so far.

## 2.9 (Optional Case Study) Thinking About Objects: Examining the Problem Statement

- Emphasize object-oriented programming (OOP)
- Object-oriented design (OOD) implementation
  - Chapters 3 to 13, 15, 22
  - Appendices G, H, I

## 2.9 (Optional Case Study) Thinking About Objects: Examining the Problem Statement

- Program Goal
  - Software simulator application
  - 2-floor elevator simulator
    - Models actual elevator operation
  - Elevator graphics displayed to user
  - Graphical user interface (GUI)
    - User can control elevator

47

## 2.9 (Optional Case Study) Thinking About Objects: Examining the Problem Statement

- Elevator Simulation
  - Model people using elevator
  - Elevator door, floor door, elevator button, floor button, elevator shaft, bell, floor, backgrounds
    - Operate accordingly or by request to avoid "injuring" person and make useless operations
  - Create person objects
  - Simulation rules
    - Elevator visits floor which  person requests for elevator service
    - One person per elevator
    - 5 seconds to move from floors

48

## 2.9 (Optional Case Study) Thinking About Objects: Examining the Problem Statement

- Application GUI
  - **First Floor**/**Second Floor** buttons create person on respective floors
    - Disable button if floor occupied by a person already
    - Unlimited number of passenger creations
  - Animation requirements
    - Passenger walking and pressing floor button
    - Elevator moving, doors opening and closing
    - Illumination of elevator lights and buttons during operation
  - Incorporating sounds
    - Footsteps when person walks
    - Button pressing clicks
    - Elevator bell rings upon elevator arrival, elevator music
    - Doors creak when opening and closing

## 2.9 (Optional Case Study) Thinking About Objects: Examining the Problem Statement

- Designing elevator system
  - Specified in requirements document through OOD analysis
    - UML
    - Design used to implement Java code
  - How system should be constructed to complete tasks
- System Structure
  - System is a set of interactive components to solve problems
    - Simplified by subsystems
      - Simulator (through ch. 15), GUI (ch. 12 and 13, display (ch. 22)
  - Describes system’s objects and inter-relationships
  - System behavior describes how system changes through object interaction

## 2.9 (Optional Case Study) Thinking About Objects: Examining the Problem Statement

- UML diagram types
  - System structure
    - Class diagram (section 3.8)
      - Models classes, or “building blocks” of a system
      - Person, elevator, floor, etc.
    - Object diagrams (section 3.8)
      - Snapshot (model) of system’s objects and relationships at specific point in time
    - Component diagrams (section 13.17)
      - Model components such as graphics resources and class packages that make up the system
    - Deployment diagrams (not discussed)
      - Model hardware, memory and runtime resources

## 2.9 (Optional Case Study) Thinking About Objects: Examining the Problem Statement

- System behavior
  - Statechart diagrams (section 5.11)
    - Model how object changes state
      - Condition/behavior of an object at a specific time
  - Activity diagrams (section 5.11)
    - Flowchart modeling order and actions performed by object
  - Collaboration diagrams (section 7.10)
    - Emphasize what interactions occur
  - Sequence diagrams (section 15.12)
    - Emphasize when interactions occur
  - Use-case diagrams (section 12.16)
    - Represent interaction between user and system
      - Clicking elevator button