SFWR ENG/COMP SCI 2S03 Principles of Programming

Dr. Ridha Khedri

Department of Computing and Software, McMaster University
Canada L8S 4L7, Hamilton, Ontario

Acknowledgments: Material based on *Java actually: A Comprehensive Primer in Programming* (Chapter 11)

Topics Covered

1. Introduction and Learning Objectives
2. File handling
   - Data records
3. Text files
   - Writing to text files
   - Excp. hand. when writing to a text file
   - Reading from text files
   - Excp. hand. reading from a text file
4. Simple GUI dialog design
   - Ending programs that use GUIs
   - Message dialogs
   - Input dialogs
   - Confirmation dialogs

- A program must communicate with its environment
- So far, we used the keyboard (as a source) and the terminal window (as destination)
- Data can be written to and read from text files
- Data can be exchanged with users using graphical dialog boxes
- After the program terminates, files can be used for keep data available
- A data file offers persistent storage (both a source and a destination for data)

Learning Objectives

- Organizing values in data records for storing in text files
- Writing string representations of primitive values and objects to text files
- Reading characters from text files and converting them to appropriate values
- Designing simple GUI dialogs using the *JOptionPane* class
- Creating message dialog boxes to present information to and require from the user

- A file refers to a specific storage area on media where data can be stored
- Data in a file is' stored as a sequence of bytes (i.e., 8 bits)
- However, it can be interpreted in different ways during reading and writing
- If the data in a file is interpreted as a sequence of bytes, the file is called a binary file
- If the data is interpreted as a sequence of characters, the file is called a text file

- In general the way in which primitive data values are stored is platform- specific
- $\rightsquigarrow$ The interpretation of a binary file changes with the platform (NOT portable)
- Java solves the problem of portability of binary files by defining the size of primitive data types
- A binary file containing such values will be interpreted correctly by a Java program on any platform

Primitive data type	Size in bytes
boolean	1
char	2
int	4
long	8
float	4
double	8

- A char value in a Java program always occupies two bytes
- HOWEVER, it might not be the case when the character is stored in a text file
  - It depends on the character encoding used to store the characters in the file
- Java provides classes that facilitate reading and writing characters from text files
  - They take into account the encoding used

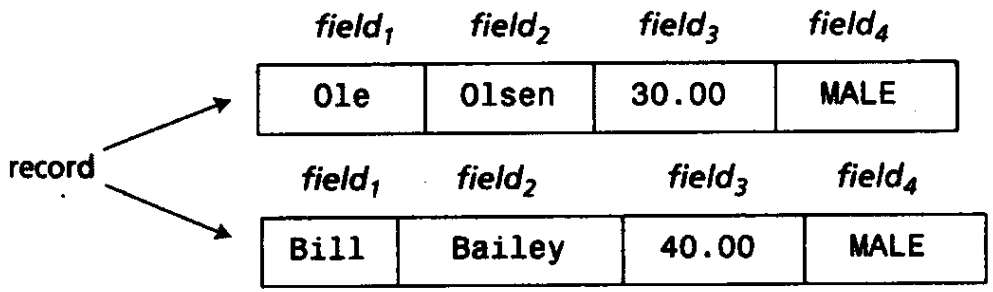
- **Problem statement:** We wants to store information about employees in a text file

- We would like to store the following information about an employee in a text file:

```
String firstName;       // Field variable 1
String lastName;        // Field variable 2
double hourlyRate;      // Field variable 3
Gender gender;          // Field variable 4
```

- It is quite common to store information as data records

- A record consists of one or more data fields (i.e., n-tuple)



```
1 // Constants to represent gender
  enum Gender {FEMALE, MALE};
```

```
  // Class representing an employee
2 class Employee {                                                    //
      (1)

4   // Static variable
    final static double NORMAL_WORKWEEK = 37.5;
6
    // Field variables
8   String firstName;
    String lastName;
10  double hourlyRate;
    Gender gender;
12
    // Constructors
14  Employee() { }
    Employee(String firstName, String lastName,
16           double hourlyRate, Gender gender) {
      this.firstName  = firstName;
18    this.lastName   = lastName;
      this.hourlyRate = hourlyRate;
20    this.gender     = gender;
    }
22
    // Determines whether an employee is female.
24  boolean isFemale() { return (this.gender == Gender.FEMALE); }

26  // Computes the salary of an employee, based on the number of hours
    // worked during the week.
28  double computeSalary(double numOfHours) {
      assert numOfHours >= 0 : "Number of hours must be >= 0";
30    double weeklySalary = hourlyRate * Employee.NORMAL_WORKWEEK;
      if (numOfHours > Employee.NORMAL_WORKWEEK) {
32      weeklySalary += 2.0 * hourlyRate * (numOfHours - NORMAL_WORKWEEK);
      }
34    return weeklySalary;
    }
36
    // Return string representation of the field values of an employee.
38  public String toString() {
      return String.format(
40      "First name: %-6s Last name: %-8s Hourly rate: %6.2f Gender: %-6s"
            ,
        this.firstName, this.lastName, this.hourlyRate, this.gender);
42  }
}
```

```
1  // Personnel register of employees
   class PersonnelRegister {                                    //
       (2)
3
     final static int MAX_NUM_EMPLOYEES = 100;
5
     // Fields
7    Employee[] employees;
     int        numOfEmployees;
9    int        numOfFemales;

11   // Default constructor
     PersonnelRegister() {
13     employees = new Employee[MAX_NUM_EMPLOYEES];
     }
15
     // Create a personnel register from an array of employees.
17   PersonnelRegister(Employee[] suppliedEmployeeArray) {
       employees = new Employee[MAX_NUM_EMPLOYEES];
19     for( int i = 0; i < suppliedEmployeeArray.length; i++) {
         registerEmployee(suppliedEmployeeArray[i]);
21     }
     }
23
     // Create a personnel register with field values from parameters.
25   PersonnelRegister(Employee[] employees, int numOfEmployees,
                       int numOfFemales) {
27     this.employees = employees;
       this.numOfEmployees = numOfEmployees;
29     this.numOfFemales = numOfFemales;
     }
31
     // Register an employee in the employee array.
33   void registerEmployee(Employee newEmployee) {
       assert numOfEmployees != employees.length:
35            "No room for more employees.";
       employees[numOfEmployees] = newEmployee;
37     numOfEmployees++;
       if (newEmployee.gender == Gender.FEMALE) {
39       numOfFemales++;
       }
41   }

43   // Continues on next slide ..............
```

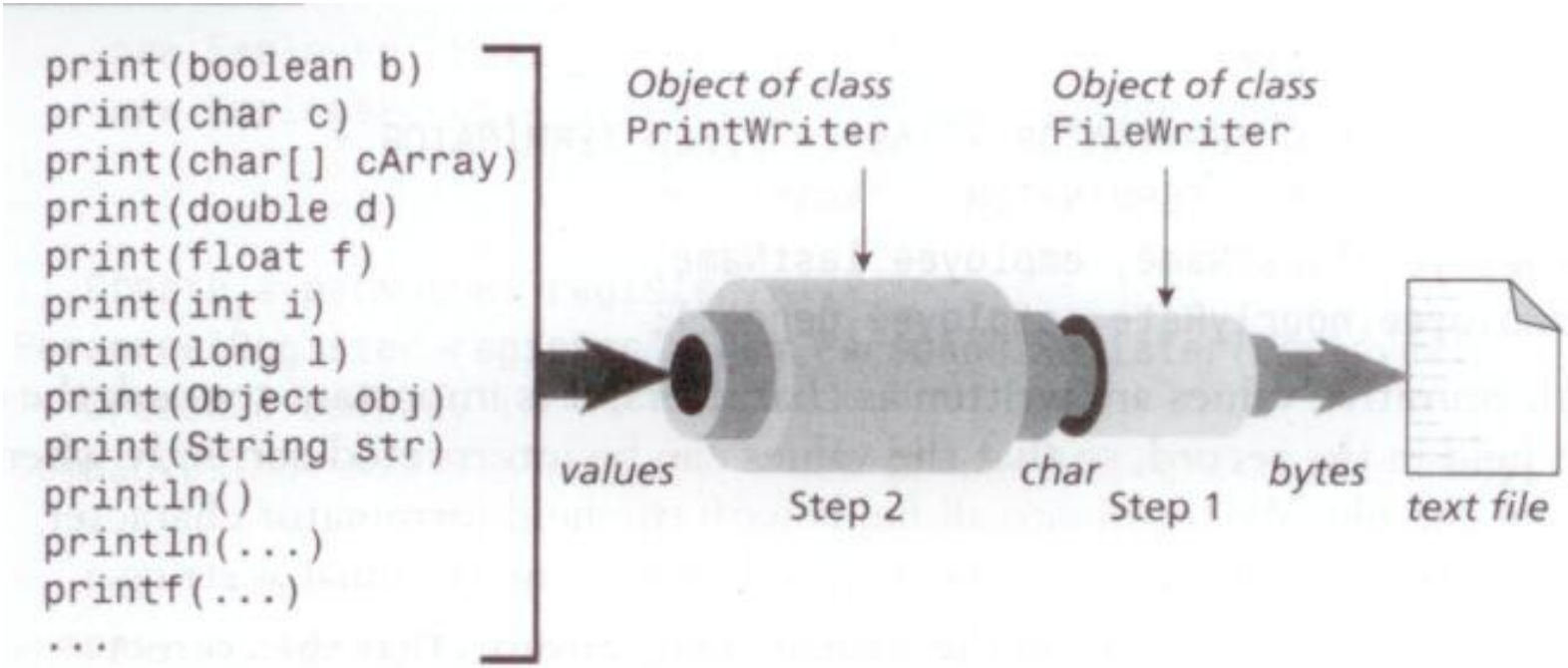
```
// ... Continuing previous slide

  // Selectors
  int getNumOfEmployees()        { return numOfEmployees; }
  int getNumOfFemales()          { return numOfFemales; }
  Employee[] getEmployeeArray() { return employees; }

  // Return employee with specified index.
  Employee getEmployee(int index) {
    assert 0 <= index && index <= numOfEmployees :
           "Index not valid";
    return employees[index];
  }

  // Replace an employee at index with another employee.
  void replaceEmployee(int index, Employee newEmployee) {
    assert 0 <= index && index <= numOfEmployees :
           "Index not valid";
    employees[index] = newEmployee;
  }

  // Compute percentage of females in the company.
  double getFemalePercentage() {
    assert numOfEmployees > 0 : "Personnel register is empty.";
    return 100.0 * numOfFemales / numOfEmployees;
  }

  // Returns statistics about the company.
  public String toString() {
    return String.format("The company has %d employees," +
                         " where %.2f%% are women.",
                          getNumOfEmployees(), getFemalePercentage());
  }
}
```

- A text file contains lines of text
- Line = sequence of characters that is terminated by a line terminator string
- The line terminator string is platform-specific (e.g., windows: "\r\n", Unix: "\n")
- The methods for file handling in Java ensure that it is interpreted correctly
- **java.io** package provides support for reading/writing data from various sources/destinations

- There are four steps to file handling:
  (1) Open the file: create a connection between the program and the file
  (2) Choose the appropriate class to convert the values: depending on whether we are reading or writing
  (3) Writing to /reading from the file
  (4) Close the file: we are finished with it $\rightsquigarrow$ thereby freeing any resources that were used

We need the following classes to write to a text file:

- The class **java.io.FileWriter**: opens the file + ensures that the characters are in the platform encoding
- The class **java.io.PrintWriter**: provides methods for converting values to their respective string representations




```
FileWriter textFileWriter = new FileWriter(textFileName); // Step 1
PrintWriter textWriter = new PrintWriter(textFileWriter); // Step 2
```

Writing: How the 4 steps for file handling apply?

(1) To open the file for writing: Create a FileWriter object

```
FileWriter textFileWriter = new FileWriter(dataFileName);
```

- If a file with the designated name does not exist, a new file with this name is created
- If the file with the designated name exists, it is reset (i.e., old content gone)
- There is another constructor that allows appending content

```
FileWriter textFileWriter = new FileWriter(dataFileName, append);
```

- the parameter append is true, this constructor will open the file for appending (Otherwise, writing starts from the beginning)

(2) Create a PrintWriter object that is connected to the FileWriter

```
PrintWriter textWriter = new PrintWriter(textFileWriter);
```

- The PrintWriter delivers characters to the FileWriter
- The FileWriter ensures that the characters it receives are correctly stored as bytes in the file
- There is another constructor that accepts a file name as parameter:

```
PrintWriter textWriter = new PrintWriter(dataFileName);
```

* creates the underlying file writer
* creates the file if necessary (always opens it for writing from the beginning)

(3) Write text representations of values

- writing uses the print methods of the PrintWriter class
- Example:

```
textWriter.printf(
    "%s" + FIELD_TERMINATOR + "%s" + FIELD_TERMINATOR +
    "%.2f" + FIELD_TERMINATOR + "%s%n",
    employee.firstName, employee.lastName,
    employee.hourlyRate, employee.gender);
```

- It is important to mark the end of each field in the record (to be able to read them later)
- We terminate all fields with the field terminator character FIELD_TERMINATOR (comma (,) $\rightsquigarrow$ comma-separated value (CSV) file format)
- Last field is terminated by the line terminator string %n

(4) Closing the file

- It is done by calling the close() method of the PrintWriter

```
textWriter.close();
```

- Calling the close () method is **strongly recommended**
  - ensures that the connection between the program and the file closed
  - ensures that any resources that were used for handling the file are freed
  - ensures that no data is lost

```
import java.io.IOException;

public class CompanyAdmin {

  public static void main(String[] args) throws IOException {      // (1)
    // Create an array of employees.
    Employee[] employeeInfo = {
        new Employee("Ole",  "Olsen",  30.00, Gender.MALE),
        new Employee("Bill", "Bailey", 40.00, Gender.MALE),
        new Employee("Liv",  "Larsen", 50.00, Gender.FEMALE)
    };

    // Create a personnel register.
    PersonnelRegister register = new PersonnelRegister(employeeInfo);
    // Create a company.
    CompanyUsingTextFiles company = new CompanyUsingTextFiles(register);

    // Print employee info in personnel register to a text file.
    company.writeAllEmployeesToTextFile("employeeFile.txt");      // (2)

    // Read employee info from a text file.
    System.out.println("Read from text file.");
    company.readAllEmployeesFromTextFile("employeeFile.txt");      // (3)

    // Print employee info to the terminal window.
    company.printAllEmployeesToTerminalWindow();
    company.printReport();
  }
}
```

```
1  import java.io.BufferedReader;
   import java.io.FileReader;
3  import java.io.FileWriter;
   import java.io.IOException;
5  import java.io.PrintWriter;

7  class CompanyUsingTextFiles {

9    final static char FIELD_TERMINATOR     = ',';  // Comma

11   // Field
     private PersonnelRegister register;
13
     // Constructors
15   CompanyUsingTextFiles() {
       register = new PersonnelRegister();
17   }

19   CompanyUsingTextFiles(PersonnelRegister register) {
       this.register = register;
21   }

23   // Print statistics
     void printReport() {
25     System.out.println(register);
     }
27
     void printAllEmployeesToTerminalWindow() {
29     Employee[] employees = register.getEmployeeArray();
       int numOfEmployees = register.getNumOfEmployees();
31     System.out.println("Number of employees: " + numOfEmployees);
       for (int i = 0; i < numOfEmployees; i++) {
33       System.out.println(employees[i]);
       }
35   }

37
// CONTINUED ON NEXT SLIDE .............
```

```
// ..... Continued from previous slide
2
  void writeAllEmployeesToTextFile(String dataFileName)
4       throws IOException {                                          //
            (4)

6     FileWriter textFileWriter = new FileWriter(dataFileName);       //
          (5)
      PrintWriter textWriter = new PrintWriter(textFileWriter);       //
          (6)
8
      // Write the number of employees in the register.
10    int numOfEmployees = register.getNumOfEmployees();
      textWriter.println(numOfEmployees);                             //
          (7)
12
      // Write info about each employee.
14    Employee[] employees = register.getEmployeeArray();
      for (int i = 0; i < numOfEmployees; i++) {
16      writeEmployeeData(textWriter, employees[i]);                  //
            (8)
      }
18
      textWriter.close();                                             //
          (9)
20  }

22  void writeEmployeeData(PrintWriter textWriter,
                           Employee employee) {                       //
                               (10)
24    // Fields separated by a field terminator character.
      textWriter.printf(
26        "%s" + FIELD_TERMINATOR + "%s" + FIELD_TERMINATOR +
          "%.2f" + FIELD_TERMINATOR + "%s%n",
28        employee.firstName, employee.lastName,
          employee.hourlyRate, employee.gender);
30  }

32
// ... CONTINUES on next slide ....
```

```
3
Ole,Olsen,30.00,MALE
Bill,Bailey,40.00,MALE
Liv,Larsen,50.00,FEMALE
```

- Writing to a file exceptions cannot be ignored (i.e., checked exception)
- The method that writes is declared with a throws clause in its header:

```
void writeAllEmployeesToTextFile(String dataFileName)
      throws IOException {                                    // (4)
  ...
  FileWriter textFileWriter = new FileWriter(dataFileName);   // (5)
  ...
}
```

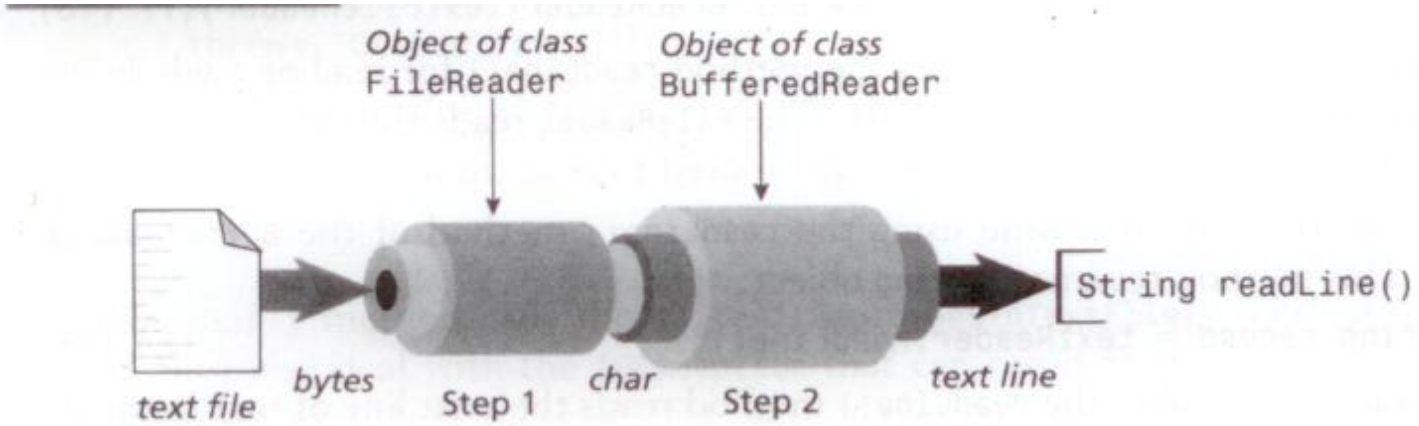
- The caller method must also handle the IOException

```
public static void main(String[] args) throws IOException {    // (1)
  ...
  company.writeAllEmployeesToTextFile("employeeFile.txt");      // (2)
  ...
}
```

- NO throws clauses results in a compile time error

We need the following classes to read from a text file

- The class java.io.FileReader
  - opens the file
  - ensures that the bytes in the file are interpreted correctly as characters
- The class java.io.BufferedReader
  - provides the ability to read a whole line of text
  - It uses a buffer
    - it is temporary storage media
    - allows to store characters temporarily in memory
    - move several characters at a time from the external media to the buffer
    - enhances performance (minimizes the number of accesses necessary)




```
FileReader     textFileReader = new FileReader(textFileName);        // Step 1
BufferedReader textReader     = new BufferedReader(textFileReader); // Step 2
```

The procedure for reading text from files

(1) Create a FileReader object to open the file for reading

```
FileReader textFileReader = new FileReader(dataFileName);
```

- The constructor accepts the name of the file to open
- If a file with the designated name does not exist, an exception is thrown
- Reading starts at the beginning of this file
- An IOException is thrown, if for some reason the file cannot be opened

(2) Create a BufferedReader object that is connected to the FileReader

```
BufferedReader textReader = new BufferedReader(textFileReader);
```

- BufferedReader class provides the method readLine( ) for reading a whole line of text
- The FileReader reads the bytes in the file as characters
- The BufferedReader can be used to read lines of text

(3) Read text lines using the readLine( ) method of the BufferedReader class

- Example:

```
String record = textReader.readLine( );
```

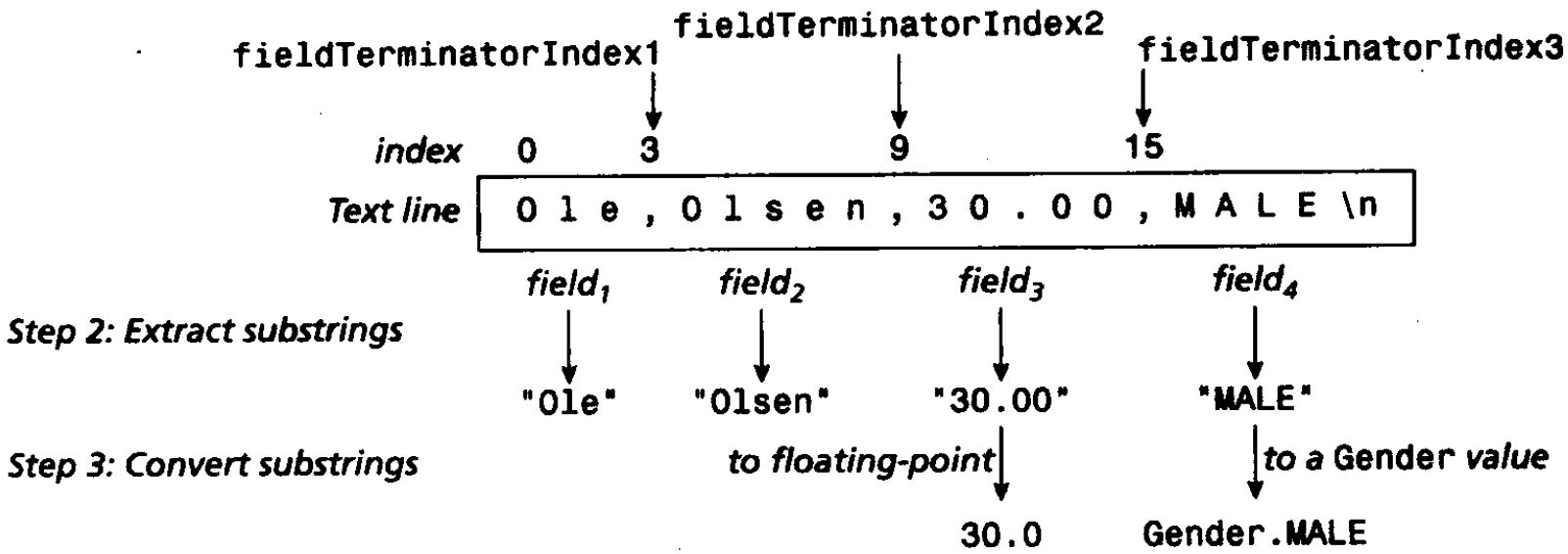
- Each successive call to the readLine( ) method reads the next line
- The readLine( ) method returns a null when there are no more characters to read(i.e., EOF)
- The line terminator string at the end of each text line is NOT a part of the returned string
- It cannot read from the file $\rightsquigarrow$ readLine( ) throws an IOException

(3) Continued .

- How to extract the characters that comprise each field value from the record?
  - Find the index of this character in the record
  - Extract the substring that comprises the characters in the field
- If the field value is NOT string value, we must convert the substring to the corresponding value

(4) Close the file

```
textReader.close();
```



```
3
Ole,Olsen,30.00,MALE
Bill,Bailey,40.00,MALE
Liv,Larsen,50.00,FEMALE
```

```
1  // ..... Continued from previous slide
3    void readAllEmployeesFromTextFile(String dataFileName)
         throws IOException {                                    //
             (11)
5
       FileReader textFileReader = new FileReader(dataFileName);  //
           (12)
7      BufferedReader textReader = new BufferedReader(textFileReader);//
           (13)
9      // Read how many employees are in the text file.
       String firstLine = textReader.readLine();                 //
           (14)
11     int totalNumOfEmployees = Integer.parseInt(firstLine);
13     // Create a personnel register.
       register = new PersonnelRegister();
15
       // Read info about all employees in the file.
17     for (int i = 0; i < totalNumOfEmployees; i++) {
         // Read an employee.
19       Employee employee = readEmployeeData(textReader);       //
             (15)
         // Register the employee.
21       register.registerEmployee(employee);
       }
23
       textReader.close();                                       //
           (16)
25   }
27 // ... CONTINUES on next slide ....
```

SFWR ENG/COMP SCI 2S03 Principles of Programming

Dr. R. Khedri

```
1  // ..... Continued from previous slide
2
3
4    Employee readEmployeeData(BufferedReader textReader)
5            throws IOException {                                    //
                 (17)
6      // Read a record, i.e. a text line.
7      String record = textReader.readLine();                        //
           (18)
8
9      // Find the index of the field terminator characters in the record.
       int fieldTerminatorIndex1 = record.indexOf(FIELD_TERMINATOR); //
           (19)
11     int fieldTerminatorIndex2 = record.indexOf(FIELD_TERMINATOR,
                                                  fieldTerminatorIndex1 + 1);
13     int fieldTerminatorIndex3 = record.indexOf(FIELD_TERMINATOR,
                                                  fieldTerminatorIndex2 + 1);
15
16     // Extract the values of the fields from the data record.
17     String firstName = record.substring(0, fieldTerminatorIndex1); //
           (20)
       String lastName = record.substring(fieldTerminatorIndex1 + 1,
19                                        fieldTerminatorIndex2);
20
21     String doubleStr = record.substring(fieldTerminatorIndex2 + 1,
                                           fieldTerminatorIndex3);
23     double hourlyRate  = Double.parseDouble(doubleStr);           //
           (21)
24
25     String genderStr = record.substring(fieldTerminatorIndex3 + 1);
       Gender gender;
27     if (genderStr.equals(Gender.MALE.toString())) {               //
           (22)
         gender = Gender.MALE;
29     } else {
         gender = Gender.FEMALE;
31     }
32
33     return new Employee(firstName, lastName, hourlyRate, gender);
34   }
35 }
```

```
1 Read from text file.
  Number of employees: 3
3 First name: Ole     Last name: Olsen     Hourly rate:  30.00 Gender: MALE
  First name: Bill    Last name: Bailey    Hourly rate:  40.00 Gender: MALE
5 First name: Liv     Last name: Larsen    Hourly rate:  50.00 Gender:
      FEMALE
  The company has 3 employees, where 33.33% are women.
```

- Exception handling for reading is analogous to that for writing to a text file
- Calls to the method readLine( ) can throw an IOException
- The calling method is declared with a throws clause

```
Employee readEmployeeData(BufferedReader textReader)
        throws IOException {                                    // (17)
  ...
  String record = textReader.readLine();                        // (18)
  ...
}
```

- The method below in the execution stack, must be declared with a throws clause

- The java.swing.JOptionPane class is useful for creating simple graphical user interfaces (GUIs)
- How this class can be used for
  - presenting information to the user
  - entering input required by the program
  - requesting the user to confirm information
- The JOptionPane class defines three **static** methods to create dialog boxes

- The JOptionPane class methods:
  - showTypeDialog
  - showMessageDialog
  - showInputDialog (See textbook for details)
  - showConfirmDialog
- These methods have many parameters in common
- Some of the parameters need not be specified (default values are used)
- All these dialog boxes are modal (all user input is directed to them)

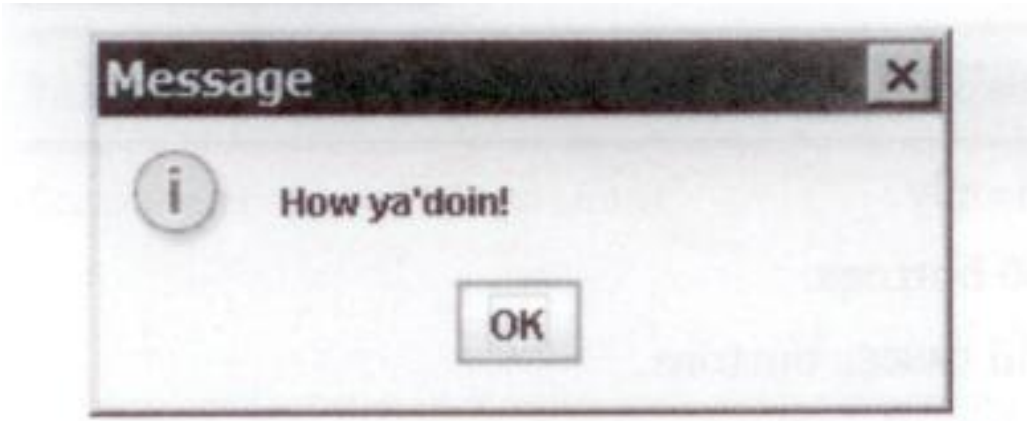
Ending programs that use GUIs

- A program usually ends when the main() method has finished executing
- When using GUI components, there is a thread is added
- Its role is to monitor the interaction between the program and the GUI
- As the last statement in the main () method, the thread must be stopped explicitly

System.exit(0) ;

- The method requires an integer value as parameter (e.g., 0 things are fine, non zero otherwise)

```
import javax.swing.JOptionPane;

public class MessageDialog {
  public static void main(String[] args) {

    JOptionPane.showMessageDialog(                    // (1)
        null,                                         // No parent window
        "How ya'doin!"                                // Message
    );

    JOptionPane.showMessageDialog(                    // (2)
        null,                                         // No parent window
        "You have hit the jackpot!",                  // Message
        "Important Message",                          // Title in the window
        JOptionPane.WARNING_MESSAGE                   // Message type
    );

    System.exit(0);                                 // (3) Terminate the
        program.
  }
}
```

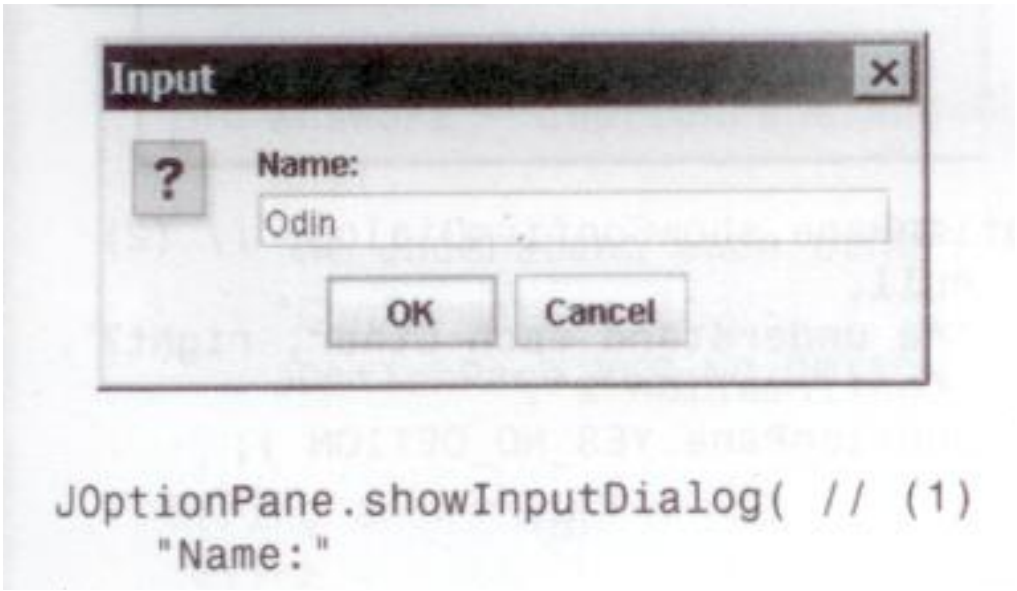


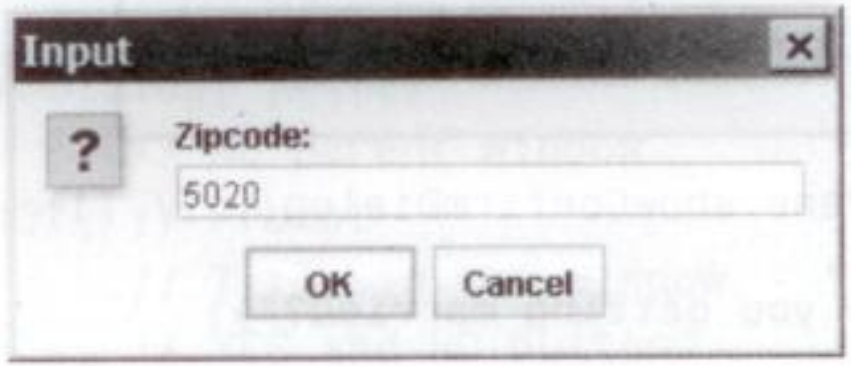

```
JOptionPane.showMessageDialog( // (1)
    null,
    "How ya'doin!");
```


```
JOptionPane.showMessageDialog( // (2)
    null,
    "You have hit the jackpot!",
    "Important Message",
    JOptionPane.WARNING_MESSAGE);
```

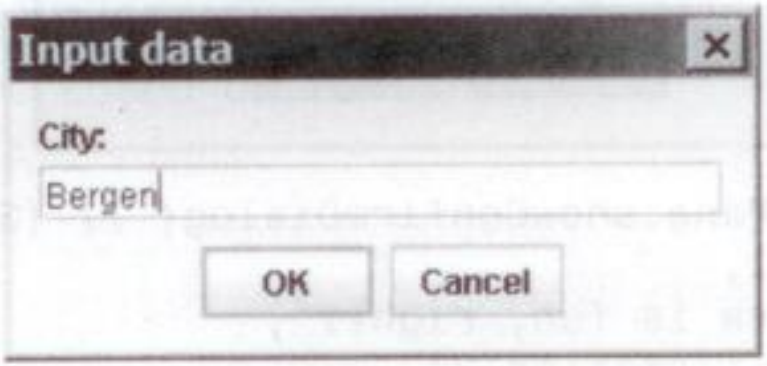
```
import javax.swing.JOptionPane;

public class InputDialog {
  public static void main(String[] args) {

    String name = JOptionPane.showInputDialog(          // (1)
        "Name:"                                         // Prompt
    );

    String zipcodeStr = JOptionPane.showInputDialog(    // (2)
        null,                                           // No parent window
        "Zipcode:"                                      // Prompt
    );
    int zipcode = Integer.parseInt(zipcodeStr);         // (3)

    String city = JOptionPane.showInputDialog(          // (4)
        null,                                           // No parent window
        "City:",                                        // Prompt
        "Input data",                                   // Title in the
            window
        JOptionPane.PLAIN_MESSAGE                       // Message type
    );

    JOptionPane.showMessageDialog(                      // (5) Message
        dialogue
        null,
        name + "\n" + zipcode + " " + city,
        "Information",
        JOptionPane.PLAIN_MESSAGE
    );

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

Using **showInputOialog( )**



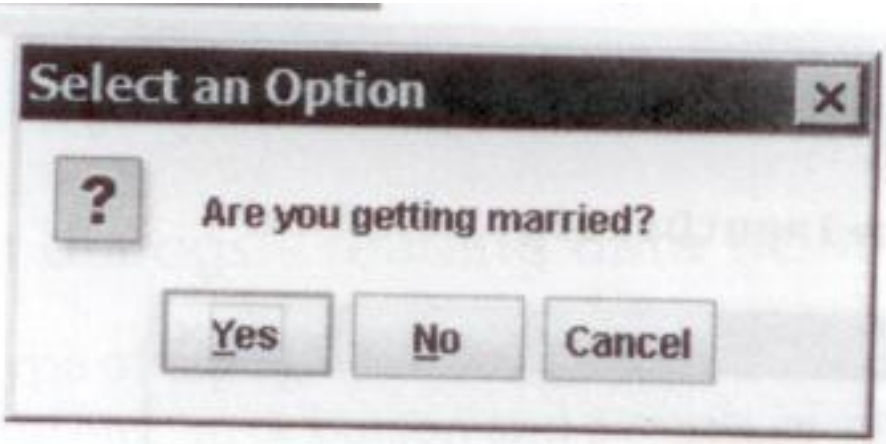



```
JOptionPane.showInputDialog( // (2)
    null,
    "Zipcode:"
);
```

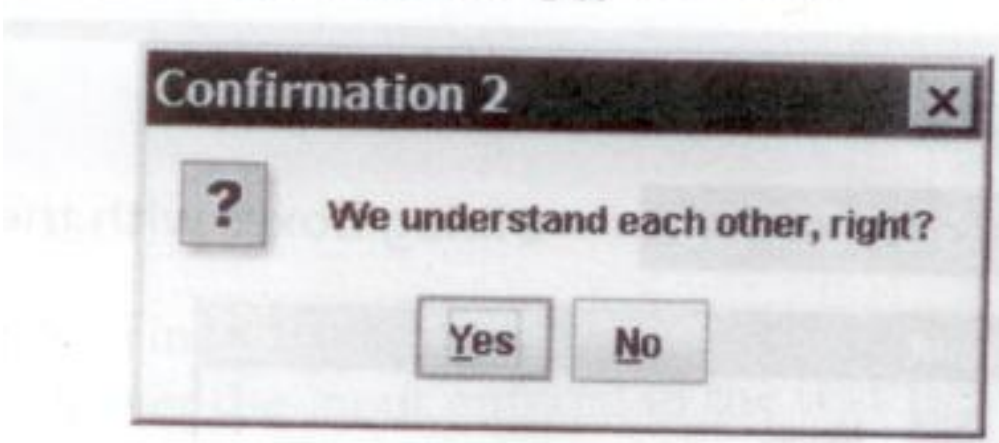



```
JOptionPane.showInputDialog(      // (4)
    null, "City:",
    "Input data", JOptionPane.PLAIN_MESSAGE);
```

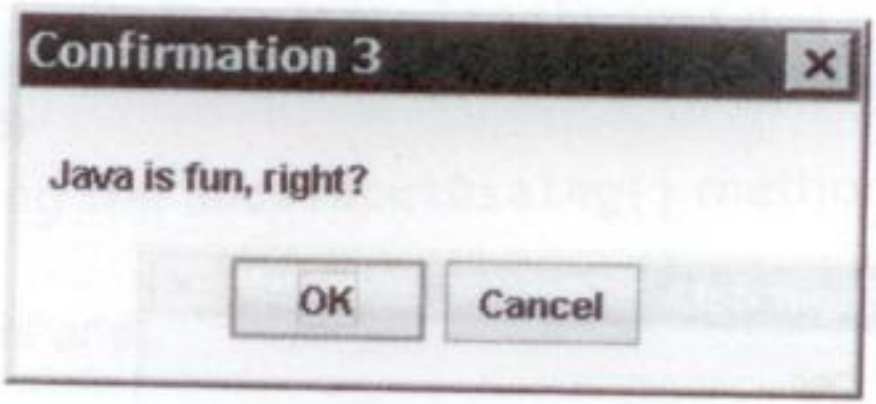
Using **showConfirmOialog( )**




```
JOptionPane.showConfirmDialog(   // (1)
    null,
    "Are you getting married?");
```




```
JOptionPane.showConfirmDialog( // (2)
    null,
    "We understand each other, right?",
    "Confirmation 2",
    JOptionPane.YES_NO_OPTION );
```




```
JOptionPane.showConfirmDialog( // (3)
    null,
    "Java is fun, right?",
    "Confirmation 3",
    JOptionPane.OK_CANCEL_OPTION,
    JOptionPane.PLAIN_MESSAGE);
```

```
import javax.swing.JOptionPane;

public class ConfirmDialog {
  public static void main(String[] args) {

    int answer1 = JOptionPane.showConfirmDialog( // (1)
        null,                                    // No parent window
        "Are you getting married?"               // Prompt
    );                                           // YES, NO and CANCEL
         buttons
    String answerStr1 = null;
    switch (answer1) {
      case JOptionPane.YES_OPTION:
        answerStr1 = "Congratulations!";
        break;
      case JOptionPane.NO_OPTION:
        answerStr1 = "All right.";
        break;
      case JOptionPane.CANCEL_OPTION:
      case JOptionPane.CLOSED_OPTION:
        answerStr1 = "Sorry I asked.";
        break;
      default:
        assert false;
    }
    JOptionPane.showMessageDialog(null, answerStr1);

    int answer2 = JOptionPane.showConfirmDialog( // (2)
        null,                                    // No parent window
        "We understand each other, right?", // Prompt
        "Confirmation 2",                        // Title in the window
        JOptionPane.YES_NO_OPTION                // YES and NO buttons
    );
    String answerStr2 = null;

// ... CONTINUED on next slide ....
```

```
1  // ... Continues from previous slide
3      switch (answer2) {
         case JOptionPane.YES_OPTION:
5          answerStr2 = "Good!";
           break;
7        case JOptionPane.NO_OPTION:
         case JOptionPane.CLOSED_OPTION:
9          answerStr2 = "All right.";
           break;
11       default:
           assert false;
13     }
       JOptionPane.showMessageDialog(null, answerStr2);
15
       int answer3 = JOptionPane.showConfirmDialog(     // (3)
17         null,                                  // No parent window
           "Java is fun, right?",                 // Prompt
19         "Confirmation 3",                      // Title in the window
           JOptionPane.OK_CANCEL_OPTION,          // OK and CANCEL buttons
21         JOptionPane.PLAIN_MESSAGE              // Message type
       );
23     String answerStr3 = null;
       switch (answer3) {
25       case JOptionPane.OK_OPTION:
           answerStr3 = "We agree!";
27         break;
         case JOptionPane.CANCEL_OPTION:
29       case JOptionPane.CLOSED_OPTION:
           answerStr3 = "Pity you won't confirm.";
31         break;
         default:
33         assert false;
       }
35     JOptionPane.showMessageDialog(null, answerStr3);

37     System.exit(0);
     }
39 }
```