Java's Object Orientation, I/O

Course Topics

Elements of the Java Platform

The Java Language

Java Arrays, Objects, Methods

▶ *Java's Object Orientation and I/O*

Interfaces, Graphical User Interfaces, and Applets

Topics this week:

- Last Week
- Methods
- Variables (Fields)
- Overloading
- Objects
- Garbage Collection
- Accessability
- Key aspects of Object Oriented Programming
- Encapsulation
- Inheritance
- File I/O
- Simple File I/O - Readers / Writers
- Attaching / Stacking Streams
- File Input Example - with Exception handling
- Assignment for next time

Last Week

Java Arrays, Objects, Methods

Arrays

- Are themselves *first class* objects
- Are **declared**, **instantiated**, and **initialized**
- Can be made of any primitive type or Class
- Have an associated field (arrayName.length) that specifies the number of elements the array can hold

Constructors

- Purpose is to guarantee object instances are initialized correctly
- Look sort of like instance methods
- Have same name as class; don't have a return type
- Are "called" using *new*

Classes - Templates for Objects

> ***Definition:*** *A class is a blueprint or prototype that defines the variables and methods common to all objects of a certain kind.*

Objects - Instances of classes

Definition: *An object is a software bundle of variables and related methods.*

Methods

Instance Methods

Instance methods are only called *through* an object: an instance of some class created by invoking *new* on a class.

```
someInstance.instanceMethod()
```

```
/** An example of a class defining counter objects.
 **/

public class Counter {

  private int count = 0;                // instance variable

  public void increment() {             // instance method
    count++;
  }

  public int getCount() {               // another
    return count;
  }

  // for stand-alone testing
  public static void main(String[] args) {

    Counter[] cArray = new Counter[5];
    for (int i = 0; i < cArray.length; i++ ) {
      cArray[i] = new Counter();
      for (int num = 0; num < 10; num++) {
        cArray[i].increment();
      }
    }

    for (int i = 0; i < cArray.length; i++) {
      System.out.println("cArray[" + i + "] = " + cArray[i].getCount());
    }
    // What is output from the above?
  }
}
```

Class methods

Utility methods associated with an entire class, not any one particular instance of the class.

```
ClassName.classMethod()
```

Qualifying a method with the keyword **static** makes that method a *class* method.

```
/** An example of a class that defines several class methods.
 **/

public class ClassCounter {

  private static int count = 0;         // a class variable

  public static void increment() {      // class method
    count++;
  }

  public static int getCount() {        // another
    return count;
  }

  public static void main(String argv[]) {

    // The normal way to call class methods, from within the class
    // or from some other class is:
    //  Classname.methodname( [arguments if any] );

    // Even if I make instances of the class, and call the static
    // methods through the instance references, the results are
    // not going to work the same way that instance methods would.

    ClassCounter[] cArray = new ClassCounter[5];
    for (int i = 0; i < cArray.length; i++ ) {
      cArray[i] = new ClassCounter();
      for (int num = 0; num < 10; num++) {
        cArray[i].increment();
      }
    }

    for (int i = 0; i < cArray.length; i++) {
      System.out.println("cArray[" + i + "] = " + cArray[i].getCount());
    }
    // What is output from the above?
  }
}
```

If you find yourself using a lot of class methods in Java, it is a sign that you are thinking in the old, procedural programming style, not like an object oriented programmer.

Variables (Fields)

Instance variables

In a good object oriented Java class, just about *all* variables ought to be instance variables.

Each object (instance) will have its own copy of these fields

Class variables

Qualifying a variable (field) with the keyword **static** makes that variable a *class* variable:

- There is only one of those variables no matter how many instances of the class are instantiated.
- All instances *share* that class variable
- There are only two good reasons to declare a class variable in Java:
  - As a `static final` constant
  - As a `private static` internal item of information that is purposely shared among object instances

Local Variables

Variables you declare and use only with a method or a smaller block of code.

```
/** SimpleBankAccount.java - illustrate instance and local cariables
 **/

public class SimpleBankAccount {

  // Instance variables

  private String id;                    // these are all "instance" variables
  private String name;                  // of associated person
  private String address;               // to send statements to
  private long   balance;

  public SimpleBankAccount(String id, String name, String address) {
    // "this" refers to "this object" or "this instance of the class"
    this.id = id;
    this.name = name;
    this.address = address;
    balance = 0;                        // same as this.balance = 0;
  }

  public String getName() {
    return name;
  }

  public void addToBalance (long amount) {
    balance = balance + amount;
  }

  public void calculateInterest () {
    double interest;                     // local variable
    interest = balance * interestRate;
    balance = balance + (long) interest; // "casting"
  }

  public float getBalance() {
    return balance;
  }

  private static double interestRate = .06; // class variable

  static void setInterestRate(double newRate) {
    interestRate = newRate;
  }
}
```

Overloading

- Two or more methods may have the same *name* but take different sets of *arguments*; they have different *signatures*.
- This is called *overloading*.
- Method signatures must differ by the type and/or number of arguments, *not* only by the data type they return;
- A common use of overloading in Java is to have multiple constructors, each taking a different set of arguments.

Question: *Why would this be useful?*

Objects

Objects are created by applying the **new** operator to a class, naming one of the class' *constructors*

You can (generally) create as many object instances of a class as you need. (Constrained only by the memory available to the JVM.)

The reference (handle) to an instance of an object that has been created needs to be saved somewhere:

- in a variable,
- in an array,
- in some other object,
- in some collection of objects.

Otherwise the instance of an object will eventually get reclaimed by Java's built-in *garbage collection* mechanism.

Garbage Collection

Fundamental to the safety and robustness of Java

All Java Objects are automatically kept track of by the Java Virtual Machine.

When no more *live* references exist, the object or array memory is made available to be garbage collected.

Makes programming Java a lot easier and less error prone.

Compared to other programming languages that lack Garbage Collection.

Garbage collection in Java happens automatically: the JVM makes unused memory available for later re-use use by the JVM.

Accessability

The creator of a Java class controls what access objects *outside* the class have to the implementation (the inner details) of objects of her class by giving variables and methods accessability qualifiers.

- `public`
  - All outside objects can call public methods.
  - Any outside object can potentially change public variables.
- `private`
  - methods are only callable within the instance methods of the class - not by subclasses.
  - variables are only accessable within the methods of the class - not from subclasses.
- `protected`
  - methods are only callable from the methods of the class and any sub classes.
  - variables are accessable within the instance methods of the class and any sub classes.
- "Package access" - the default if not otherwise specified:
  - Instances of any class in the same package may call methods with *package* access.
  - Instances of any class in the same package can access *package* variables

Question: *Why add an accessability qualifier to a method or variable?*

Key aspects of Object Oriented Programming

Encapsulation

A class should contain all the logic necessary to define how objects of the class work and no more.

Inheritance

In Java one class **extends** another class. In doing so the derived class should do something in addition, or in a different way, from the parent (super) class.

Polymorphism

Objects can expose different sets of capabilities at different times during a programs execution.

In Java this is done by extending other classes and by implementing various interfaces.

Encapsulation

- supports *modularity* - an object can be written and maintained independently of other objects.
- *information hiding* - objects have public methods that they expose for other objects to use. These methods permit other objects to send "messages" to it. Private variables and methods of the object are implementation details that can be changed at any time without affecting other objects.
- Classes should expose to the outside just the methods needed to make the object do the things it is designed to do;
- None of the internal details related to the particular implementation of the class should be visible to the outside.

```
/** A class that encapsulates information about and messages to
 ** send to Employee objects.
 **/
public class Employee {

  public static void main (String [] args) {
    // create a single instance of the Employee class,
    Employee anEmp = new Employee("Joe Student");

    // assign a name and a salary value (any value), using the constructor
    // and the setSalary() methods
    anEmp.setSalary(50000.);

    // Access the current salary value (using the getSalary() method)
    double current = anEmp.getSalary();

    // and increase the Employee's salary by 10% using the
    // setSalary method again.
    current = current + (current * .1);
    anEmp.setSalary(current);

    // then have the Employee object identify itself and it's
    // current salary value.
    System.out.println(anEmp);
  }

  private String name;
  private double salary;
  private Employee boss;

  /** a public constructor for the class that will insure that Employee
      objects always have a name field
  **/
  public Employee( String empName) {
    name = empName;
  }

  public String getName() {
    return name;
  }

  /** a method to set an Employee object's salary field
   **/
  public void setSalary( double value ) {
    salary = value;
  }

  /** a method to get an Employee object's current salary field  value
   **/
  public double getSalary( ) {
    return salary;
  }

  /** a method to set an Employee object's field referencing another
      Employee object
  **/
  public void setBoss ( Employee boss ) {
    this.boss = boss;
  }

  /** a method to return a String identifying the field contents of an
      Employee
  **/
  public String toString() {
    return name + " salary:" + salary;
  }
}
```

Inheritance

You say one class **extends** another when you want to *re-use* most of the capabilities of a class, but want to add some additional capabilities, over-ride some, or provide other specialization.

```
/** encapsulate the characteristics of any one student
 ** @author: Jonathan Doughty
 **/

public abstract class AnyStudent {

  private static long nextId = 0;

  // Instance variables (fields) that will be associated with each student

  private String name;
  private long   id;
  private long   tuitionPaid;
  private long   tuitionOwed;

  // A constructor that will create student objects with specified name and id
  public AnyStudent( String studentName) {
    name = studentName;
    id = nextId;
    nextId++;
  }

  // I'm not going to allow other users of the AnyStudent class create
  // Students without a name and an id value.
  private AnyStudent() {
  }

  public String toString() {
    return name + ":" + id;
  }

  public void payTuition(int amount) {
    tuitionPaid += amount;
    tuitionOwed -= amount;
  }

  public long tuitionOwed() {
    return tuitionOwed;
  }
}
```

```
/** encapsulate the characteristics of an UnderGrad student
 ** @author: Jonathan Doughty
 **/

public class UnderGradStudent extends AnyStudent {

  private float GPA;

  // A constructor that will create student objects with specified name and id
  public UnderGradStudent( String studentName) {
    super(studentName);                 // invoke AnyStudent constructor
  }

  public float getGPA() {
    return GPA;
  }

  public String toString() {
    String basic = super.toString();
    return basic + " GPA:" + GPA;
  }

  // Other methods associated with being an undergraduate student ...
}
```

```
/** encapsulate the characteristics of a Grad student
 ** @author: Jonathan Doughty
 **/

public class GradStudent extends AnyStudent {

  private Faculty advisor;
  private Course  teachingAssistantFor;

  // A constructor that will create student objects with specified name and id
  public GradStudent( String studentName) {
    super(studentName);
  }

  public void writeThesis() {
    // mumble
  }

  public String toString() {
    String basic = super.toString();
    return basic + " grad";
  }

  // Other methods associated with being a grad student ...
}
```

File I/O

- Data stored in fields only lasts while a method is executing; perhaps for the length of time that an application is running.
- Just about any program in any language, to be useful, is going to have to get input from somewhere, process that input in some way, and store the result somewhere.
- Most programs will operate by getting data from a file, somewhere, and/or producing output in a file.

Simple File I/O - Readers / Writers

```
/** adapted from JavaTutorial: example of simple file I/O using
 ** classes derived from Reader and Writer.
 **/

import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopyChars {

  public static void main(String[] args) {
    if (args.length != 1)
      System.out.println("usage: CopyChars file_to_copy");
    else {
      CopyChars obj = new CopyChars();
      try {
        obj.makeCopy(args[0]);
      }
      catch (IOException e) {
        System.out.println(e);
      }
    }
  }

  public void makeCopy(String filename) throws IOException {
    FileReader in = new FileReader(filename);
    FileWriter out = new FileWriter(filename + ".copy");

    int c;

    // Read characters from the reader until in.read returns -1
    while (true) {
      c = in.read();
      if (c == -1)
        break;
      out.write(c);
    }

    in.close();
    out.close();
  }
}
```

Attaching / Stacking Streams

Reading/writing files a character at a time is inefficient.

An easy way to improve performance: Use Java’s capability to attach one stream to another.

```
/** Copy file contents using buffering
 **/

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class BufferedCopy {
  public static void main(String[] args) {
    if (args.length != 1)
      System.out.println("usage: BufferedCopy file_to_copy");
    else {
      BufferedCopy obj = new BufferedCopy();
      try {
        obj.makeCopy(args[0]);
      }
      catch (IOException e) {
        System.out.println(e);
      }
    }
  }

  public void makeCopy(String filename) throws IOException {
    FileReader fr = new FileReader(filename);
    BufferedReader bufIn = new BufferedReader(fr);

    FileWriter fw = new FileWriter(filename + ".copy");
    BufferedWriter bufOut = new BufferedWriter(fw);

    // An alternative way to write the previous four lines is:
    // BufferedReader bufIn = new BufferedReader( new FileReader(args[0]));
    // BufferedWriter bufOut = new BufferedWriter( new FileWriter(filename + "copy));
    // since fr and fw are only used in the BufferedReader/Writer
    // constructors.

    String s;

    // Read and write lines of text (through buffers)
    while (true) {
      s = bufIn.readLine();             // trims end of line character
      if (s == null)
        break;
      bufOut.write(s);
      bufOut.newLine();                 // adds an end of line character
    }

    // Flush anything remaining and release resouurces
    bufIn.close();
    bufOut.close();
    bufIn = null;
    bufOut = null;
  }
}
```

Copy of a ~ 1/2 megabyte file :
CopyChars - 42 seconds
BufferedCopy - < 8 seconds

Buffered input also enables some other capabilities that would not be possible otherwise, e.g., *readLine()*.

Question: *Why isn't readLine() implemented for other Stream and Reader classes?*

Other Stream and Reader/Writer classes provide other useful filters.

Question: *What's the point of the alternative way of creating in and out?*

SavitchIn.java

The textbook (and the accompanying CD) includes a utility class called *SavitchIn* that provides methods to access data and interpret it in a variety of ways. It only works from console input, however.

InputExample.java

```java
/** InputExample - demonstrate interpreting numeric values from
 ** input from console (System.in)
 **/

import java.io.IOException;

public class InputExample {

  public static void main( String[] args) {
    InputExample example = new InputExample();

    System.out.print("enter int values, end with end of file character:");
    System.out.flush();

    int sum = example.sumIntValues();
    System.out.println("Sum is " + sum);
  }

  public int sumIntValues() {
    byte input[] = new byte[80];
    int bytesRead ;
    String inputString = null;
    String trimmedString = null;
    int intValue = 0;
    int sum = 0;

    try {
      while (true) {
        // Read input bytes and create a String object from them
        bytesRead = System.in.read(input);
        if (bytesRead < 0)   // read returns -1 of no more is available
          break;

        inputString = new String(input, 0, bytesRead);

        // Remove any whitespace from either end
        trimmedString = inputString.trim();

        // After removing whitespace check if the string has zero length
        if (trimmedString.length() == 0)
          continue;                    // try again

        // Try to convert what is left to an integer value
        // Note that this doesn't check for non-digits

        intValue = Integer.parseInt(trimmedString);
        sum += intValue;
      }
    }
    catch (IOException e) {
      System.out.println("whoops, caught IO Exception");
    }
    return sum;
  }
}
```

Question: *What happens if the user enters non-digit characters?*

File Input Example - with Exception handling

```
/** InputInterpreter - demonstrate interpreting numeric values from
 ** input from console (System.in) or a file.  Also uses
 ** overloaded constructors and exception handling.
 **/

// Classes other than those in java.lang used:
import java.io.IOException;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.InputStreamReader;

/** A class that demonstrates reading and interpreting input from
 ** the console or a file.
 **
 ** @author Jonathan Doughty
 **/
public class InputInterpreter {

  BufferedReader input = null;
  boolean consoleInput = false;

  /** Constructs an object that will read lines for interpretation
   ** from System.in: the keyboard generally.
   **/
  public InputInterpreter() {
    input = new BufferedReader( new InputStreamReader( System.in ));
    consoleInput = true;
  }

  /** Constructs an object that will read lines for interpretation
   ** from a file with the indicated name.
   **/
  public InputInterpreter(String fileName)
    throws FileNotFoundException
  {
    input = new BufferedReader( new FileReader( fileName ));
  }

  /** Tell whether input is being read from console.
   **/
  public boolean isConsole() {
    return consoleInput;
  }

  /** Reads a String value from the next line of input,
   ** trimming blanks from either end, and ignoring blank lines.
   **
   ** @return value interpreted from input
   **/
  public String readString() throws EOFException {

    String result = null;

    try {
      while (true) {
        // Read a line of from input source
        result = input.readLine();

        // Check if a line was sucessfully read
        if (result == null)
          throw new EOFException("end of input");

        // Remove any whitespace from either end
        result = result.trim();

        // Check if the string has zero length
        if (result.length() == 0)
          continue;                     // try again
        else {
          break;                        // from infinite loop
        }
      }
    }
    catch (EOFException eof) {
        throw eof;                      // re-throw it
    }
    catch (IOException ie) {
      // Ignore any other IOExceptions
      // (EOFException is a sub-class)
    }
    return result;
  }

  /** Reads an int value from the next line of input, ignores
   ** all errors.
   **
   ** @return value interpreted from input
   **/
  public int readIntValue() throws EOFException {

    int result = 0;

    try {

      String line = readString();
      result = Integer.parseInt(line);
    }
    catch (EOFException eof) {
        throw eof;                      // re-throw it
    }
    catch (NumberFormatException ne) {
      // ignore input that isn't legal
    }
    return result;
  }

  /** Reads a double value from the next line of input, ignores
   ** all errors.
   **
   ** @return value interpreted from input
   **/
  public double readDoubleValue() throws EOFException {

    double result = Double.NaN;   // Initialize to "Not a Number"

    try {
      String line = readString();
      Double wrapper = Double.valueOf(line);
      result = wrapper.doubleValue();
    }
    catch (EOFException eof) {
        throw eof;                      // re-throw it
    }
    catch (NumberFormatException ne) {
      // ignore input that isn't legal
    }
    return result;
  }
}
```

TestInterpretInput.java

... something to test it with

```
/** TestInterpretInput operation by reading one of each value
 ** supported by its public interface either from the console
 ** or a file indicated on the command line.
 **/

import java.io.FileNotFoundException;
import java.io.EOFException;
import java.io.IOException;

public class TestInputInterpreter {

  InputInterpreter inputSource = null;

  /** Create an InputInterpreter associated with either the
   ** console or a file whose name is given in the srgument array.
   **/
  private void setUpInput(String args[]) throws FileNotFoundException {
    if (args.length == 0) {

      // Read and interpret console input
      inputSource = new InputInterpreter();

    }
    else {

      // Read and interpret input from file indicated in args
      inputSource = new InputInterpreter(args[0]);
    }
  }

  /** Read and interpret the contents of the input source.
   **/
  private void readInput() {

    boolean doPrompts = inputSource.isConsole();

    try {
      while (true) {
        if (doPrompts)
          System.out.println("Enter a string, e.g., your name");

        // Declare a String and then ask inputSource to read a String
        // and assign it to the String object you've declared



        if (doPrompts)
          System.out.println("Enter an int, e.g., your age");

        // Declare an int and then ask inputSource to read an int
        // value and assign it to the int variable you've declared



        if (doPrompts)
          System.out.println("Enter digits for a double, e.g., taxes paid");

        // Declare a double and then ask inputSource to read a
        // double value and assign it to the double variable
        // you've declared



        // Report the String, int and double values you've read in



      }
    }
    catch (EOFException eof) {
      System.out.println("caught end of file");
    }
  }


  // Read records from the indicated file

  public int readRecords( String filename ) {

    int numRecords = 0;
    int recordsRead = 0;

    try {
      // Read and interpret input from file indicated in filename
      inputSource = new InputInterpreter(filename);

      // Assume the input file consists of an initial line that has
      // the number of records, followed by the records themselves

      numRecords = inputSource.readIntValue();

      // Now read that many records.  Each record consists of a
      // String, an int value, and a double value, each on
      // separate lines.  So the file contains
      // string
      // int
      // double
      // string
      // int
      // double
      // ...

      String stringVal;
      int    intVal;
      double dblVal;
      for (recordsRead = 0; recordsRead < numRecords; recordsRead++ ) {

        stringVal = inputSource.readString();
        intVal = inputSource.readIntValue();
        dblVal = inputSource.readDoubleValue();

        // do something useful with the three input items read,
        // like print them, make a new object with them ...
      }
    }
    catch (FileNotFoundException e) {
      // whoops
      return 0;                           // read 0 records
    }
    catch (EOFException e) {
      // got to end of file earlier than expected
      System.err.println("End of file after " + recordsRead + " records");
      return recordsRead;
    }
    catch (IOException e) {
      // some other unexpected problem; have it report itself
      System.err.println(e);
      return 0;
    }
    return numRecords;
  }

  public static void main( String[] args) {

    TestInputInterpreter tester = new TestInputInterpreter();

    try {
      // One of the constructors can throw a FileNotFound
      // exception; thus the need to enclose the following in a
      // try block.
      tester.setUpInput(args);
      tester.readInput();
    }
    catch (FileNotFoundException e) {
      System.out.println(e);
    }
  }
}
```

Assignment for next time

Reading

- Chapter 7 - Event Driven Programming Using the AWT

Homework

Goal:

- Add input/output capabilities to one of your classes;
- read data, process it, and produce some results calculated from the input using multiple classes.

Purpose:

- To access data from an external source,
- To learn how to instruct the computer to process it,
- How to break data into pieces associated with objects.
- Learn about I/O principles;
- Building more complex programs by combining simple pieces.

  Using the Employee class you wrote for the second assignment (or using the Employee class on the next page), create a new class called Company which will have the following methods related to calculating a simple payroll for company employees.

  -

```
public Employee inputEmployeeRecord()
```

This instance method should read information for a *single* Employee from an input source and create a new Employee object, returning the single Employee as the result.

Employee records being read from an input file will consist of a name on one line, and an annual salary figure on the next line:

You might want to make use of the InputInterpreter class.

○

```
public void inputFile(String filename)
```

This method will open the file specified by the filename argument and use the inputEmployeeRecord method to read and create Employee objects. The resulting Employees should be stored in an instance array of Employees for later processing.

Use the file of employee input data that is given on the next page.

For input, you may use either SavitchIn, InputInterpreter, or your own input/output and data interpretation capabilities.

If you use SavitchIn (which only reads from System.in (normally the keyboard), you can use *input re-direction* on both Windows and Unix:

```
java MyClass <employees.dat
```

though you'll have to figure out some way of handling how many employees are being entered or detecting the end of input. You may find it easier to use the InputInterpreter class. *You must use the records provided. If you modify the input file in any way, please submit your modified input file too.*

- ```
  public void processPayroll()
  ```

  This method should access each Employee object in the array that you populated in the inputFile() method. Use the Employee class' getSalary() method to access the employee's annual salary. Use the Employee class' getName() method to access the Employee's name. Calculate a biweekly salary amount (annual salary divided by 26 pay periods) and then report as output the employee's name and biweekly salary.

- ```
  public static void main(String[] args)
  ```

  Have your Company class's main method make an instance of the Company class, then call inputFile with the name of the employee data file on that Company instance and finally call the Company instance's processPayroll() method.

Be sure to document your programs.

Hand in listings of your class source files, and a transcript of its execution.

Employee.java

```
/** A class that encapsulates information about and messages to
 ** send to Employee objects.
 **/
public class Employee {

  public static void main (String [] args) {
    // create a single instance of the Employee class,
    Employee anEmp = new Employee("Joe Student");

    // assign a name and a salary value (any value), using the constructor
    // and the setSalary() methods
    anEmp.setSalary(50000.);

    // Access the current salary value (using the getSalary() method)
    double current = anEmp.getSalary();

    // and increase the Employee's salary by 10% using the
    // setSalary method again.
    current = current + (current * .1);
    anEmp.setSalary(current);

    // then have the Employee object identify itself and it's
    // current salary value.
    System.out.println(anEmp);
  }

  private String name;
  private double salary;
  private Employee boss;

  /** a public constructor for the class that will insure that Employee
      objects always have a name field
  **/
  public Employee( String empName) {
    name = empName;
  }

  public String getName() {
    return name;
  }

  /** a method to set an Employee object's salary field
   **/
  public void setSalary( double value ) {
    salary = value;
  }

  /** a method to get an Employee object's current salary field  value
   **/
  public double getSalary( ) {
    return salary;
  }

  /** a method to set an Employee object's field referencing another
      Employee object
  **/
  public void setBoss ( Employee boss ) {
    this.boss = boss;
  }

  /** a method to return a String identifying the field contents of an
      Employee
  **/
  public String toString() {
    return name + " salary:" + salary;
  }
}
```

employees.dat

Bill Clinton
200000
Bill Gates
2000000000
Jack Edelman
700000
Susan Koniak
88500
Sally Saltzberg
78900
Bonnie Nagel
32500
Lena Baisayev
540500
Joe Maio
58000
Craig Seymour
125800
Charles Dutton
245000
Jack Trescott
23900
Justin Blum
93700
Daniel Williams
67700
Richard Frank
90000
Kwabena Nimarko
118000