



CM1210 Object Oriented Java Programming



SESSION 3

A Class with More Classes and Reading/Writing Files 😊



REMEMBER FROM LAST WEEK:

Problem

- Create a phone book application that **stores names and numbers**, allows a user to **add** entries and **retrieve** the number for a given name, and **load/save** the **data** to **file**.

Classes

- PhoneBookEntry: **// COMPLETED LAST WEEK**
- PhoneBook: **// TO DO**

Finished **PhoneBookEntry** application

```
class PhoneBookEntry {  
  
    private String name;  
    private String number;  
  
    public PhoneBookEntry( String inName, String inNumber ) {  
        name = inName;  
        number = inNumber;  
    }  
  
    public String getName( ) {  
        return name;  
    }  
  
    public String getNumber( ) {  
        return number;  
    }  
  
    public void setNumber( String inNumber ) {  
        number = inNumber;  
    }  
  
    public String toString( ) {  
        String s = name + "\t(" + number + ") ";  
        return s;  
    }  
}
```

PhoneBookEntryTest application

Lets now write an application to use our new PhoneBookEntry class and test our code:

```
public class PhoneBookEntryTest {  
    public static void main( String[] args ) {  
        PhoneBookEntry e = new PhoneBookEntry( "Bob", "+44  
            0123456789" );  
        System.out.println( e ); }  
}
```

LIBRARY APPLICATION

Problem

- Create a library application that **stores titles** and **barcodes** of books, allows a user to **add** new books and **retrieve** the barcode for a given title, and **loads/saves** the **data** to **file**.

Suggested Classes

- Book
- Library

TASK: Construct the Book Class in Java and write an appropriate Class to test it.

The PhoneBook class

- **PhoneBook** class will hold instances of **PhoneBookEntry** objects, just as a real Phone Book will contain Entries.
- Lets start by storing our **PhoneBookEntry** objects in an array.

```
PhoneBookEntry[] entries = new PhoneBookEntry[10];
```

- This will create an array of type **PhoneBookEntry** of length 10.
- **But Remember:** we can't resize arrays in Java.
- Now, we don't know how many entries there will be in advance, so we're going to have to overcome this shortcoming.
- But for now, lets assume that there will be less than 10 entries and build a **PhoneBook** class.

A Basic PhoneBook class

```
public class PhoneBook {
    private PhoneBookEntry[] entries;
    private int size;
    final private static int MAX_ENTRIES = 10;

    public PhoneBook( ) {
        entries = new PhoneBookEntry[MAX_ENTRIES];
    }

    public void add( String name, String number ) {
        System.out.println("Size: " + size);
        if (size != MAX_ENTRIES) {
            entries[size] = new PhoneBookEntry( name, number );
            size++;
        }
    }

    public String toString() {
        StringBuffer temp = new StringBuffer();
        for (int i = 0; i < size; ++i) {
            temp.append( entries[i].toString() + "\n" );
        }
        return temp.toString();
    }
}
```

PhoneBookTest application

Lets now write an application to use our new PhoneBook class and test our code:

```
public class PhoneBookTest {  
  
    public static void main( String[] args ) {  
  
        // Create and fill phone book  
        PhoneBook pb = new PhoneBook();  
        pb.add( "Bob", "+44 (0) 483984" );  
        pb.add( "Carl", "38478" );  
        pb.add( "Don", "3878" );  
        pb.add( "Ed", "3848" );  
        pb.add( "Frank", "8478" );  
  
        // Output whole book  
        System.out.println();  
        System.out.println( pb );  
        System.out.println();  
    }  
}
```


Worked example

- However, this is far from ideal!!
- We don't want to have to set a fixed size for our phone book in advance.
- With a little more work we can make our storage more intelligent, by following the psuedocode below when we add an entry:

Algorithm Add a new entry:

```
if array is full then
    create new array with double the size
    copy existing entries into new array
    repoint reference to new array
end if
add entry at next free position
increment free position
```

Worked example

EXAMPLE: PhoneBook Folder

```
public class PhoneBook {

    private PhoneBookEntry[] entries;
    private int size;
    private int maxEntriesLength = 1;

    public PhoneBook( ) {
        entries = new PhoneBookEntry[maxEntriesLength];
    }

    public void add( String name, String number ) {
        System.out.println("Size: " + size + " , " + maxEntriesLength);
        if ( size == entries.length ) {
            System.out.println("Doubling");
            maxEntriesLength = 2 * maxEntriesLength;
            PhoneBookEntry[] temp = new PhoneBookEntry[ maxEntriesLength ];
            System.arraycopy( entries, 0, temp, 0, entries.length );
            entries = temp;
        }
        entries[size] = new PhoneBookEntry( name, number );
        System.out.println("Adding : " + name);
        size++;
        System.out.println("Size: " + size + " , " + maxEntriesLength);
    }

}
```

Finding Numbers

Lets implement a `numberFor` method, so we can search the number for a given name. We could use a linear search algorithm:

Linear search

Linear search is a simple search algorithm that can find the position of a specified value (called the key) within an array. The key is simply compared to each element in turn and its position is returned if it is found. If it is not found then -1 is returned.

Algorithm 2 Linear Search

```
Require: array, key  
for each element of the array do  
    if element = key then  
        return index of element  
    end if  
end for  
return -1
```

Lets try and make it easier

Easier `numberFor ()` method

Given that the **Strings** are **objects**, we can loop through the phonebook and compare the search string with each name string, using the **`equals ()`** method.

```
public String numberFor( String name ) {  
    for (int i = 0; i < size; ++i) {  
        if ( list[i].getName().equals(name) ) {  
            return list[i].getNumber();  
        }  
    }  
    return "NOT FOUND";  
}
```

Worked example

- In fact we don't need to write our own code to implement a **collection** (an object that can hold references to other objects) that can grow arbitrarily.
- The core Java API contains several classes that we could use depending on the features we would like our collection to have.
- We will start with the **Vector** class.
- From Java 1.5, the Vector class makes use of a new feature – **generics**.
- This allows us to specify the type of object that the collection will contain when we declare the Vector.
- Previously collections in Java stored elements of type Object that had to be cast when they were accessed.

Worked example

EXAMPLE: PhoneBookVector Folder

```
public class Vector<E>  
    extends AbstractList<E>  
    implements List<E>, RandomAccess, Cloneable, Serializable
```

The Vector class implements a growable array of objects. Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.

```
import java.util.Vector;  
  
public class PhoneBook {  
    private Vector<PhoneBookEntry> entries;  
  
    public PhoneBook( ) {  
        entries = new Vector<PhoneBookEntry>( );  
    }  
  
    .....  
}
```

The usual question at this stage is:

Why **Vector**?

They're really old!!

Everyone uses **ArrayLists** now

Ok, that's easy.....

Worked example

EXAMPLE: PhoneBookArrayList Folder

```
public class ArrayList<E>  
extends AbstractList<E>  
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.)

```
import java.util.ArrayList;  
  
public class PhoneBook {  
  
    private ArrayList<PhoneBookEntry> entries;  
  
    public PhoneBook( ) {  
        entries = new ArrayList<PhoneBookEntry>( );  
    }  
  
    .....  
  
}
```


Reading Text Files

Reading text files

Using the **Scanner** class

Reading from standard input

```
Scanner in = new  
Scanner(System.in);  
int x = in.nextInt();
```

Reading from input.txt file

```
Scanner in = new Scanner(new  
File("input.txt"));  
int x = in.nextInt();  
in.close();
```

NOTE: this will not compile as written — working with files in Java requires us to handle errors using **exception handling**

Exception Handling

- *Exception handling* provides a flexible mechanism for handling/recovering from errors.
- An *exception* is a problem that prevents the normal execution of a method.
- Exceptions are only intended for *exceptional* circumstances and should not be used for general flow control.

Example

```
public class ExceptionTest {  
  
    public static void main( String[] args ) {  
  
        try {  
            System.out.println(  
                "Trying Integer.parseInt( \"NOT AN INT\" )" );  
            int i = Integer.parseInt( "Not an int" );  
        }  
        catch ( Exception e ) {  
            System.out.println("Some error" + e);  
            e.printStackTrace();  
        }  
        finally {  
            System.out.println( "Finally always reached" );  
        }  
    }  
}
```

General Form

General form

```
try {  
    // Code that may throw exception  
}  
catch (ExceptionType e) {  
    // Handle exception  
}  
finally { // Optional  
    // Tidy up  
}
```

Example (Throwing exceptions)

```
if (Boolean-expression) {  
    throw new NumberFormatException("any extra information");  
}
```

Exceptions are objects of the Exception class (or a class that extends the Exception class). Useful methods defined in this class include:

- toString
- printStackTrace

Stack Traces

Definition

The Java virtual machine uses a *call stack* to track the currently executing method and the method that called the current method. In any Java application the *main* method is always the first method on the stack and the last method to leave the stack.

- When an exception is thrown, execution of the current method stops. The JVM looks for a handler in the current method
- If no handler is found, the exception is passed through the call stack until a handler is found
- The application will exit if no handler is found and information including a *stack trace* is printed

Checked Exceptions

For certain types of exception, called checked exceptions, Java insists that if a method can throw an exception, you **must** handle it.

Scanner

```
public Scanner(File source)
    throws FileNotFoundException
```

Constructs a new Scanner that produces values scanned from the specified file. Bytes from the file are converted into characters using the underlying platform's default charset.

Parameters:

source - A file to be scanned

Throws:

`FileNotFoundException` - if source is not found

Checked exceptions can be *handled* by:

- Use of try and catch blocks
- Denoting that your method may throw a checked exception:

```
public void myMethod(File f) throws FileNotFoundException {
    Scanner s = new Scanner(f);
    // Other code
}
```

Example 1 – Read and Reverse

```
import java.util.Scanner;
import java.io.File;

public class Reverse {

    public static void main( String[] args ) {

        try {

            Scanner in = new Scanner( new File(args[0]) );

            while ( in.hasNextLine() ) {
                System.out.println( new StringBuffer(

                    in.nextLine()).reverse() );
            }

            in.close();
        }
        catch ( Exception e ) {
            System.out.println( e );
        }
    }
}
```

Example 1 – Sum Doubles

```
import java.util.Scanner;
import java.io.File;

public class Sum {

    public static void main( String[] args ) {

        try {
            Scanner in = new Scanner( new File(args[0]) );

            double total = 0;
            while (in.hasNextDouble()) {
                total += in.nextDouble();
            }

            System.out.printf( "Sum is %.4f%n", total );

            in.close();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```


Worked example

Reading a phone book from text file

- We can represent a phone book in a text file by having a particular entry on each line
- Each line would contain a name and a number separated by a space

Worked example

EXAMPLE: PhoneBookLoader Folder

```
import java.util.Scanner;
import java.io.File;

public class PhoneBookLoader {

    public static PhoneBook loadPhoneBook( String filename ) throws Exception {
        File fileIn = new File( filename );

        try {
            Scanner in = new Scanner( fileIn );
            PhoneBook book = new PhoneBook();

            while( in.hasNextLine() ) {
                // Get a line of the text file
                String line = in.nextLine();

                // Separate the line into name and number
                String[] parts = line.split(",");
                String name = parts[0];
                String number = parts[1];

                // Create an entry and add it to the phone book
                book.add( name, number );
            }
            in.close();

            return book;
        }
        catch( Exception e ) {
            System.out.println( "Problem reading file: " + filename );
            throw e; // re-raise exception
        }
    }

    public static void main( String[] args ) {
        try {

            String filename = args[0];
            PhoneBook pb = loadPhoneBook( filename );
            System.out.println( pb );

        }
        catch( Exception e ) {
            // Do nothing
        }
    }
}
```

Writing files

- So we've used Scanner(File) to read text files. This is a convenience function provided by **Scanner**
- The package **java.io** provides many more classes for **reading** and **writing text** and **binary** files.
- The **Reader** and **Writer** classes and their subclasses are used to handle **text** files.
- The Reader and Writer classes are **abstract**.
- To perform **text input/output** we use one the **subclasses** of Reader or Writer

- **FileReader**: constructs a Reader that can read from a file. Only provides low-level read methods.
- **BufferedReader**: wraps around a **FileReader** and provides a high-level **readLine** method.


Text Files

ReaderTest.java
+
WriterTest.java

Use subclasses of InputStream and OutputStream classes:

```
Date d = new Date();
ObjectOutputStream out = new ObjectOutputStream( new FileOutputStream( "test.dat" ) );
out.writeObject( d );
out.close();
ObjectInputStream in = new ObjectInputStream( new FileInputStream( "test.dat" ) );
Date sc = (Date)in.readObject();
in.close();
```

Note the object read from file must be *cast* to the correct type

- 
- Java provides a mechanism, called object **serialization** where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.
 - After a serialized object has been written into a file, it can be read from the file and deserialized. That is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.
 - The JVM allows an object that has been be serialized on one Operating System to be deserialized on an entirely different one.
 - Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

ReadTest.java + WriteTest.java

Binary vs Text

Binary	Text
Compact files	Easy for user to read
Easier to code	More portable
Includes some error checking	More complex to code
Awkward if classes change	Error checking needs to be coded

LIBRARY APPLICATION

Problem

- Create a library application that **stores titles** and **barcodes** of books, allows a user to **add** new books and **retrieve** the barcode for a given title, and **loads/saves** the **data** to **file**.

Suggested Classes

- Book
- Library

TASK: Construct the Library Class in Java, that utilises your Book Class and provides the required functionality, as defined under 'Problem' above.

Questions

