Java Introduction

Part II

Outline

• The Differences between Java C++
• Inheritance
• Abstraction
• Basic input & output classes
• Tokenizing
The Differences between Java and C++

- Syntax Differences

<table>
<thead>
<tr>
<th>Operator and keyword overloading</th>
<th>Stand-alone functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header files</td>
<td>#typedef or #define</td>
</tr>
<tr>
<td>Pointers</td>
<td>Structs or unions</td>
</tr>
<tr>
<td>Multiples inheritance</td>
<td>goto statement</td>
</tr>
</tbody>
</table>

- String Differences

In Java, Strings are objects rather than null-terminated arrays. Java Strings do not end with null character.

```c
char string[] = "hello";
int length = 0;
while (string[length] != '\0') length++;
```

This C++ code will not work as expected
• Memory Management Differences

Java eliminates the need for programmers to manage memory allocations; there are no malloc() or free() functions to worry about. Java allocates memory at run time and frees unreferenced memory with periodic garbage collection.

• Data Storage Differences

-- Java primitive data types do not vary among operating systems or hardware architectures; an int is always 32 bits, a long is always 64 bits, and floating point types are always IEEE 754 Standard. This is because the target operating system and hardware for Java code is the standard Java Virtual Machine.

-- Type casting is automatic only when no information will be lost, as in casting an int to a float.

-- A primitive data type cannot be cast to an object.
• **Array Differences**
  -- Run time checks the array indices.
  
  eg: declares type a[3](a 3-element array), in C++ you can accidentally reference the 9th element, while in Java the interpreter prevents such errors.

  -- An array is really a subclass of the Object class, which has a data member named length that can be automatically set to the number of members in the array.

```
int i;
Ball theBall[] = new Ball[3];
i = theBall.length; // sets i to 3
```

• **Inheritance**
  
  • **Definition:** A *subclass* is a class that extends another class. A subclass inherits state and behavior from all of its ancestors. The term "superclass" refers to a class's direct ancestor as well as to all of its ascendant classes.

  • Every class in Java has one and only one immediate superclass.
Members inherited by a subclass

- Superclass members declared as public or protected.
- Superclass members declared with no access specifier as long as the subclass is in the same package as the superclass.

Subclasses don't inherit a superclass's member if the subclass declares a member with the same name. In the case of member variables, the member variable in the subclass hides the one in the superclass. In the case of methods, the method in the subclass overrides the one in the superclass.

- A subclass cannot override methods that are declared final in the superclass.
- Example:
\textbf{class} Picture // Root class for our hierarchy

\{
\textbf{public} Picture () {} // Class constructor
// Methods that will be inherited by sub-classes
\textbf{public void} load (String fileName) {
    \texttt{tell("Picture", "load");}
\}
\textbf{public void} show () {
    \texttt{tell("Picture", "show");}
\}
\textbf{public void} tell (String className, String methodName) {
    \texttt{System.out.println("Class "+className+" method "+methodName+" called");}
\}
\}
\}

\textbf{class} PictureCanvas \texttt{extends} Picture {
\textbf{public} PictureCanvas () \{super();\}; //constructor
\textbf{public void} draw () \{ \texttt{tell("PictureCanvas", "draw");} \}
\}

\textbf{class} TexturedCanvas \texttt{extends} PictureCanvas // Picture grand-child class
\{
\textbf{public} TexturedCanvas () \{super();\}; //constructor
\textbf{public void} show () \{ \texttt{tell("TexturedCanvas", "show");} \}
\}
public class Sample {
public static void main(String[] args) {

// Create an object of each class in the hierarchy
Picture pic = new Picture();
PictureCanvas canvas = new PictureCanvas();
TexturedCanvas textured = new TexturedCanvas();

pic.load("File");
canvas.load("File");
textured.load("File"); //Class Picture method load called

//Class Picture method load called
//Class Picture method load called
//Class Picture method load called

picRef = pic; picRef.show(); //Class Picture method show called
picRef = canvas; picRef.show(); //Class Picture method show called
picRef = textured; picRef.show(); //Class TexturedCanvas method show called
}
}
Hiding Member Variables

- Member variables defined in the subclass hide member variables that have the same name in the superclass.

```java
class Super {
    Number aNumber;
}
class Subbie extends Super {
    Float aNumber;
    super.aNumber // Number type
}
```

Abstraction

- A subclass must override methods that are declared abstract in the superclass, or the subclass itself must be abstract.

```java
abstract class Polygon {
    abstract public double getArea();
    abstract public int getSideCount();
    public int getSideLength() {
        return sideLength;
    }
    ....
}
```
class Triangle extends Polygon { // sub-class of the Polygon class
    public Triangle() {}; // Triangle constructor
    // Define the parent class abstract methods
    public double getArea() // Get the area of the triangle {
        return (sideLength * sideLength) * Math.sqrt(3.0/4.0);
    }
    public int getSideCount() // Get the count of the sides
    { return 3; }
}

class Square extends Polygon // sub-class of the Polygon class
{
    public Square() {}; // Square constructor
    // Define the parent class abstract methods
    public double getArea() // Get the area of the square {
        return sideLength * sideLength;
    }
    public int getSideCount() // Get the count of the sides {
        return 4;
    }
}
I/O: reading & Writing

• Reading
  -- open a stream
  -- while more information
  -- read information
  -- close the stream

• Writing
  - open a stream
  - while more information
  - write information
  - close the stream
Basic input & output classes

- **Character Streams:** 16-bit Character
  - Reader
  - Writer
- **Byte Streams:** 8-bit byte
  - InputStream
  - OutputStream
- **Object Stream**
  - ObjectInputStream
  - ObjectOutputStream
InputStream Methods

- **read()** reads a single byte, an array, or a subarray of bytes. It returns the bytes read, the number of bytes read, or -1 if EOF.
- **skip()** which takes long, skips a specified number of bytes of input and return the number of bytes actually skipped.
- **available()** returns the number of bytes that can be read without blocking.
- **close()** close the input stream to free up the resources.

OutputStream methods

- **void write()** takes a byte array and writes an array or subarray of bytes.
- **void flush()** forces any buffered output to be written.
- **void close()** closes the stream and frees up system resources
**Reader**

- **read()** reads a single character and returns a character read as an integer in the range from 0 to 65535 or a -1 if the end of the stream is reached.

- **read (char[] buffer, int offset, int length)** reads characters into a portion of an array (starting at offset up to length number of characters) and returns the number of characters read or -1 if the end of the stream is reached.
Writer

- void write() takes a character and writes single character in 16 low-order bits
- void write(char[] buffer, int offset, int length) takes a character array and writes a portion of an array of characters

Sources and Sinks

- Byte arrays
- Pipes
- Sequences
- Char arrays
```java
import java.io.*;
import java.util.*;

class StringExample {
    static BufferedReader system_in = new BufferedReader (new InputStreamReader (System.in));
    public static void main(String argv[]) {
        StringWriter sw = new StringWriter();
        for (int i = 0; i < 3; i++) {
            Hotel a_hotel = new Hotel();
            a_hotel.input(system_in);
            a_hotel.write_to_string(sw);
        }
        // Print it
        String output = sw.toString();
        System.out.println("String is ");
        System.out.println(output);
    }
}

class Hotel {
    private String name;
    private int rooms;
    private String location;
    boolean input(BufferedReader in) {
        try {
            System.out.println("Name: ");
            name = in.readLine();
            System.out.println("Rooms: ");
            String temp = in.readLine();
            rooms = to_int(temp);
            System.out.println("Location: ");
            location = in.readLine();
        }
    }
```
catch(IOException e) {
    System.err.println(e);
    return false;
}

return true;
}

boolean write_to_string(StringWriter sw) {
    sw.write(name); Integer i = new Integer(rooms);
    sw.write(i.toString());
    sw.write(location);
    sw.write('/n'); return true;
}

static int to_int(String value) {
    int i = 0; try { i = Integer.parseInt(value); } catch(NumberFormatException e) {} return i;
}

catch(NumberFormatException e) {} return i; }

---

**Files**

- The File class manipulates disk files and is used to construct FileInputStreams and FileOutputStreams.

- When we construct a File object, it represents that file on disk. When we call its methods, we manipulate the underlying disk file.
File Constructors

- File(String filename)
- File(String pathname, String filename)
- File(File directory, String filename)

Test Methods

- exists(): asks if the file actually exists.
- canRead(): asks if the file is readable.
- canWrite(): asks if the file can be written to.
- isFile(): asks if it is a file (as opposed to a directory).
- isDirectory(): asks if it is a directory.
Action Methods

- `renameTo()` : renames a file or directory.
- `delete()` : deletes a file or directory.
- `mkdir()` : creates a directory specified by a File object.
- `mkdirs()` : creates all the directories and necessary parents in a File specification.

FileInputStream and FileOutputStream

- `FileInputStream` reads from a disk file. You can pass that constructor either the name of a file or a File object that represents the file. The FileInputStream object is a source of data.

- `FileOutputStream` writes to a disk file. You can pass it a File object or a name. The FileOutputStream object is a sink for data.
import java.io.*;
import java.util.*;
class FileExample {
    static BufferedReader system_in = new BufferedReader (new InputStreamReader(System.in));
    public static void main(String argv[]) {
        try {
            FileOutputStream fos = new FileOutputStream("file.dat");
            // Read in three hotels
            for (int i = 0; i < 3; i++)
                { Hotel a_hotel = new Hotel();
                  a_hotel.input(system_in);
                  a_hotel.write_to_fos(fos); }
            fos.close();
        } catch(IOException e) { System.out.println(e); }
        // Now display it
        byte [] buffer = null;
        File a_file = new File("file.dat");
        System.out.println("Length is " + a_file.length());
        System.out.println(" Can read " + a_file.canRead());
        try {
            FileInputStream fis = new FileInputStream(a_file);
            int length = (int) a_file.length();
            buffer = new byte[length];
            fis.read(buffer); fis.close(); }
        catch(IOException e) { System.out.println(e); }
        String s = new String(buffer);
        System.out.println("Buffer is" + s);}}
Tokenizing

- StringTokenizer class
- StreamTokenizer class
- tokens & delimiters

abc|def?ghi

First Token
Second Token
Third Token

StringTokenizer

- StringTokenizer(String string_to_tokenize, String delimiters)
- boolean hasMoreTokens()
- String nextToken()
- String nextToken(String new_delimiters)
- int countTokens()
import java.io.*;
import java.util.*;
public class StringTokenizerExample
{
    public static void main(String args[])
    {
        String line = "abc|def?ghi";
        StringTokenizer st = new StringTokenizer(line, "|?");
        while (st.hasMoreTokens())
        {
            String s = st.nextToken();
            System.out.println("Token is " + s);
        }
    }
}
End of Java (II)

Thank you!