

CSC 551: Web Programming

Fall 2001

Java Overview

- Design goals & features
 - platform independence, portable, secure, simple, object-oriented, ...
- Language overview
 - program structure, public vs. private
 - instance variables/methods vs. class variables/methods
 - primitive vs. reference types
 - libraries, APIs
 - class definitions, inheritance, wrapper classes
 - linked structures

Java

Java was developed at Sun Microsystems, 1995

- originally designed for small, embedded systems in electronic appliances
- initial attempts used C++, but frustration at limitations/pitfalls

recall: C++ = C + OOP features

the desire for backward compatibility led to the retention of many bad features

desired features (from the Java white paper):

simple	object-oriented	robust
platform independent	architecture neutral	portable
dynamic	interpreted	high-performance
distributed	multi-threaded	secure

note: these are desirable features for any modern language

thus, Java has become very popular, especially when Internet related

also, Sun distributes free compilers (JDK) and open source

Language features

simple

- syntax is based on C++ (familiarity → easier transition for programmers)
- removed many confusing and/or rarely-used features
e.g., explicit pointers, operator overloading, automatic coercions
- added memory management (reference count/garbage collection hybrid)

object-oriented

- OOP facilities similar C++, all methods are dynamically bound
- pure OOP – everything is a class, no independent functions*

robust

- lack of pointers and memory management avoids many headaches/errors
- libraries of useful, tested classes increases level of abstraction
 - arrays & strings are ADTs, well-defined interfaces

Language features (cont.)

platform independence

- want to be able to run Java code on multiple platforms
- neutrality is achieved by mixing compilation & interpretation
 1. Java programs are translated into *byte code* by a Java compiler
 - byte code is a generic machine code
 2. byte code is then executed by an interpreter (Java Virtual Machine)
 - must have a byte code interpreter for each hardware platform
- an Applet is a special form of Java application
 - byte code is downloaded with page, JVM is embedded in browser

architecture-neutral

- no implementation dependent features
 - e.g., sizes of primitive types is set (unlike C++)

portable

- byte code will run on any version of the Java Virtual Machine (JVM)

Language features (cont.)

dynamic

- JVM links classes at run-time as they are needed
- if supporting class is recompiled, don't have to recompile entire project

interpreted

- needed for platform independence
- interpreted → faster code-test-debug cycle, better run-time error checking

high-performance

- faster than traditional interpretation since byte code is "close" to native code
- still somewhat slower than a compiled language (e.g., C++)

Language features (cont.)

distributed

- extensive libraries for coping with TCP/IP protocols like HTTP & FTP
- Java applications can access remote URL's the same as local files

multi-threaded

- a *thread* is like a separate program, executing concurrently
- can write Java programs that deal with many tasks at once by defining multiple threads (same shared memory, but semi-independent execution)
- threads are important for multi-media, Web applications

secure

- Java applications do not have direct access to memory locations
 - memory accesses are virtual, mapped by JVM to physical locations
 - downloaded applets cannot open, read, or write local files
- JVM also verifies authenticity of classes as they are loaded
- *Sun claim: execution model enables virus-free*, tamper-free* systems*

Java for the Web

Java applets provide for client-side programming

- unlike JavaScript, Java is full-featured with extensive library support
- Java and its APIs have become industry standards
 - the language definition is controlled by Sun, ensures compatibility
 - Applications Programming Interfaces standardize the behavior of useful classes and libraries of routines
- support for applets is provided by both IE and Netscape

Java servlets provide similar capabilities on the server-side

- alternative to CGI programs, more fully integrated into Web server

First Java program

```
/**
 * Hello World Application: prints "Hello world!" message on the screen.
 * @author Dave Reed
 * @version 10/10/01
 */

public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello world!"); // prints message
    }
}
```

comments in Java

// and /* . . . */ work the same as in C++

/** . . . */ designate documentation comments

- Visual J++ displays documentation comments in side window for quick reference
- can be used to automatically generate HTML documentation (javadoc)

First Java program (cont.)

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello world!"); // prints message
    }
}
```

class definitions in Java

- similar to C++ (but no semi-colon at end)
can contain data fields (instance variables) & member functions (methods)
precede class definition with *public* to make available to all programs
- there are no stand-alone functions in Java
but can have a class with a static main method, automatically called
- must be stored in a file of same name with .java extension
e.g., `HelloWorld.java`

First Java program (cont.)

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello world!"); // prints message
    }
}
```

accessibility in Java

- **public** same as C++, accessible to all
- **protected** accessible to class, derived classes, & classes in same package
- **package-only** accessible to class and all classes in same package **DEFAULT**
- **private** same as C++, accessible only to methods in the class

main must be public to be executed

First Java program (cont.)

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello world!"); // prints message
    }
}
```

class vs. instance variables

- as in C++, by default each object has its own copies of data fields thus, known as *instance variables*
- as in C++, a variables declared *static* are shared by all class objects thus, known as *class variables*
- similarly, can have a static method (*class method*) can only operate on class variables, accessible from the class itself

```
class Math
{
    public static final double PI = 3.14159; // access as Math.PI
    public static double sqrt(double num) { . . . } // access as in Math.sqrt(9.0)
    . . .
}
```

First Java program (cont.)

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello world!"); // prints message
    }
}
```

command line arguments

- main has a parameter (an array) for accessing the command line

```
System.out.println("Hello " + args[0]);
```

output

- **System** class that contains various system routines/classes
- **out** class that contains output routines
- **println** method for displaying text to the screen

Example: character input

```
public class Count
{
    public static void main(String[] args) throws java.io.IOException
    {
        int count = 0;

        while (System.in.read() != -1) {
            count++;
        }
        System.out.println("Input has " + count + " chars.");
    }
}
```

text input is awkward in Java

`System.in.read()` reads a character (more general console I/O libraries exist)

primitive types: same as C++, but sizes are set

byte (8 bits) char (16 bits) short (16 bits) int (32 bits) long (64 bits)
float (32 bits) double (64 bits) boolean

arithmetic & relational operators: same as C++

control structures: same as C++, but no goto

Example: arithmetic & control

```
/**
 * Simple program that prints a table of temperatures
 *
 * @author     Dave Reed
 * @version   10/10/01
 */
```

note similarities to C++ code

```
public class FahrToCelsius
{
    public static void main(String[] args)
    {
        double lower = 0.0, upper = 300.0, step = 20.0;

        System.out.println("Fahr\t\tCelsius");
        System.out.println("----\t\t-----");

        for (double fahr = lower; fahr <= upper; fahr += step) {
            double celsius = (5.0/9.0) * (fahr-32.0);
            System.out.println(fahr + "\t\t" + celsius);
        }
    }
}
```

Example: arrays and math

```
import java.util.*;
import java.lang.Math;

public class RandomTest
{
    private static final int NUM_REPS = 10000;
    private static final int MAX_RANGE = 10;

    public static void main(String[] args)
    {
        int[] nums = new int[MAX_RANGE];

        for (int i = 0; i < nums.length; i++) {
            nums[i] = 0;
        }

        Random randy = new Random();
        for (int i = 0; i < NUM_REPS; i++) {
            int index = Math.abs(randy.nextInt()) % nums.length;
            nums[index]++;
        }

        for (int i = 0; i < nums.length; i++) {
            System.out.println(i + ": " +
                (nums[i]*100.0/NUM_REPS) + "%");
        }
    }
}
```

final is the same as
const in C++

primitive types (int, double, char, ...) behave the same as in C++

- space is allocated when declaration is reached, automatically deallocated

user-defined & library types are known as *reference types*

- must explicitly allocate using new, but don't need to deallocate

Primitive vs. reference types

the distinction between primitive & reference types is important

- space for a primitive object is implicitly allocated
→ variable refers to the actual data (stored on the stack)
- space for a reference object must be explicitly allocated in the program
→ variable refers to a pointer to the data (which is stored in the heap)

*Note: unlike with C++, programmer is not responsible for deleting dynamic objects
JVM performs automatic garbage collection to reclaim unused memory*

Java only provides by-value parameter passing

- but reference objects are implemented as pointers to dynamic memory
- resulting behavior mimics by-reference

```
public void Assign(String str)
{
    str = "foo";
}
```

```
String s = "bar";
Assign(s);
System.out.println(s);
```

Java ADTs

```
import java.util.*; // needed for Random class
import java.lang.Math; // needed for Math.abs function

public class Die
{
    private int myNumSides, myNumRolls;
    private Random myRandom;

    public Die(int numSides) {
        myRandom = new Random();
        myNumSides = numSides;
        myNumRolls = 0;
    }

    public int Roll() {
        myNumRolls++;
        return (Math.abs(myRandom.nextInt()) % myNumSides) + 1;
    }

    public int NumSides() {
        return myNumSides;
    }

    public int NumRolls() {
        return myNumRolls;
    }
}
```

the class constructor:

- allocates reference types
- initializes primitive types

in Java, class definition is in one file

- no equiv. to .h & .cpp

```
import Die;

class Rollem {
    public static void main(String[] args) {
        Die die1 = new Die(6);
        Die die2 = new Die(6);
        int roll = die1.Roll() + die2.Roll();
        System.out.println("Dice total = " + roll);
    }
}
```

MarbleJar class

```
import java.util.*;
import java.lang.Math;

public class MarbleJar
{
    private int numBlack, numWhite;
    private Random randy;

    public MarbleJar(int black, int white)
    {
        randy = new Random();
        numBlack = black;
        numWhite = white;
    }

    public String DrawMarble()
    {
        if (Math.abs(randy.nextInt())%(numBlack+numWhite) < numBlack)
        {
            numBlack--;
            return "BLACK";
        }
        else {
            numWhite--;
            return "WHITE";
        }
    }

    public void AddMarble(String color)
    {
        if (color == "BLACK") {
            numBlack++;
        }
        else if (color == "WHITE") {
            numWhite++;
        }
    }

    public boolean IsEmpty()
    {
        return (numBlack + numWhite == 0);
    }
}
```

Puzzle class

```
import MarbleJar;

public class Puzzle
{
    private static final int NUM_BLACK = 10;
    private static final int NUM_WHITE = 7;

    public static void main(String[] args)
    {
        MarbleJar jar = new MarbleJar(NUM_BLACK, NUM_WHITE);

        while (!jar.isEmpty()) {
            String marble = jar.DrawMarble();
            if (jar.isEmpty()) {
                System.out.println("Only one marble left -- it is " +
                    marble + ".");
            }
            else {
                String marble2 = jar.DrawMarble();
                System.out.print("I drew " + marble + " and " + marble2);

                if (marble == marble2) {
                    System.out.println(" -- adding a BLACK marble.");
                    jar.AddMarble("BLACK");
                }
                else {
                    System.out.println(" -- adding a WHITE marble.");
                    jar.AddMarble("WHITE");
                }
            }
        }
    }
}
```

Java libraries

String class (automatically loaded from java.lang)

```
int length()
char charAt(index)
int indexOf(substring)
String substring(start, end)
String toUpperCase()
boolean equals(Object)
...
```

Array class (automatically loaded from java.lang)

```
int length      instance variable
char [](index) operator
String toString()
boolean equals(Object)
...
```

Java provides extensive libraries of data structures & algorithms

```
java.util →      Vector      Stack      LinkedList
                  Dictionary  HashTable  Random
                  Calendar
```

Java & inheritance

Java does NOT have templates

- generic functions/classes are obtained via *inheritance*
 - with inheritance, can derive a new class from an existing class
 - automatically inherit attributes & methods of the parent class
 - object of derived class can be used wherever parent object is expected
- every reference type is implicitly derived from the `Object` class
- by defining data structures that contain `Objects`, any reference type can be stored (and even mixed)

```
Stack things = new Stack();

String str = "foobar";
things.push(str); // pushes String as an Object

Calendar today = Calendar.getDate(); // pushes Calendar as an Object
things.push(today);
```

Java & inheritance (cont.)

to display an object using `System.out.print`, that object must have a `toString` method that returns the corresponding output string

method calls are dynamically bound in Java

- an object knows its own type, so method calls find the appropriate code

```
while ( !things.empty() ) {
    Object top = things.pop(); // removes & returns top Object
    System.out.println(top); // toString method for the appropriate
} // type is called to display
```

if desired, can coerce `Objects` back to their more specific types

```
Calendar today = (Calendar)things.pop();

String word = (String)things.pop();
```

Wrapper classes

since primitive types are different from reference types,
Java handles them very differently

- a primitive type is NOT an Object
- thus, cannot have a Vector of ints, a Stack of chars, ...

THIS SEEMS LIKE A BIG PROBLEM!

solution: "wrapper classes"

- a wrapper class defines an Object that encapsulates a primitive type

```
Character ch = new Character('A');
```

- can access the underlying primitive value using `typeValue`

```
char c = ch.charValue();
```

- thus, if you want to use a primitive where an object is expected, wrap it

```
Stack stk = new Stack();  
stk.push(new Character('A'));
```

Example: delimiter matching

each char must be wrapped in
a Character before pushing

pop returns an Object, which
must be coerced into a
Character, which can be
accessed using `charValue()`

Note: Strings are not
accessed the same as arrays

- `charAt(i)` instead of `[i]`
- `length()` instead of `length`

```
import java.io.*;  
import java.util.Stack;  
  
public class DelimMatch {  
    public static void main(String[] args) throws java.io.IOException  
    {  
        System.out.print("Enter expression: ");  
        BufferedReader d = new BufferedReader(new InputStreamReader(System.in));  
        String line = d.readLine();  
  
        Stack parens = new Stack();  
        for (int i = 0; i < line.length(); i++) {  
            if (line.charAt(i) == '(' || line.charAt(i) == '[') {  
                parens.push(new Character(line.charAt(i)));  
            }  
            else if (line.charAt(i) == ')' || line.charAt(i) == ']') {  
                if (parens.empty()) {  
                    System.out.println("Error: unmatched right delimiter at index " + i);  
                    return;  
                }  
                else {  
                    Character top = (Character)parens.pop();  
                    if (!match(top.charValue(), line.charAt(i))) {  
                        System.out.println("Error: mismatched delimiters at index " + i);  
                        return;  
                    }  
                }  
            }  
        }  
        if (!parens.empty()) {  
            System.out.println("Error: unclosed left delimiter");  
        }  
        else {  
            System.out.println("Expression is OK");  
        }  
    }  
  
    private static boolean match(char ch1, char ch2)  
    {  
        return ((ch1 == '(' && ch2 == ')') || (ch1 == '[' && ch2 == ']'));  
    }  
}
```

No pointers?

recall: there are no (explicit) pointers in Java

DOES THIS PRECLUDE LINKED STRUCTURES?

ANSWER: No

- since reference types are really pointers to structures, can use the type itself where a C++ pointer would be expected

```
public class IntNode
{
    public int data;
    public IntNode next;    // reference to another node structure
}

IntNode front = null;    // initializes front to an empty list

IntNode newNode = new IntNode();    // allocates a new node structure
newNode.data = 14;    // stores 14 in data field
newNode.next = front;    // sets next field (reference) to front

front = newNode;    // sets front reference to new node
```

Linked list class

```
public class ObjectNode
{
    public Object data;
    public ObjectNode next;
}

public class LinkedList
{
    private ObjectNode front;

    public LinkedList() {
        front = null;
    }

    public void Insert(Object obj) {
        ObjectNode newNode = new ObjectNode();
        newNode.data = obj;
        newNode.next = front;
        front = newNode;
    }

    public String toString() {
        String listStr = "[";
        ObjectNode current = front;
        while (current != null) {
            listStr += " " + current.data;
            current = current.next;
        }
        return listStr + " ]";
    }
}
```

generic linked list class

- ObjectNode can store any Object (i.e., any reference type)
- LinkedList contains a linked sequence of ObjectNodes

```
LinkedList list = new LinkedList();

for (int i = 1; i <= 10; i++) {
    list.Insert(new Integer(i));
    list.Insert("foo");
}

System.out.println(list);
```

Next week...

Java applets

- APPLET tag in HTML
- applet methods & event handling
- graphical applets

read Chapter 21

as always, be prepared for a quiz on

- today's lecture (moderately thorough)
- the reading (superficial)