

# CSC 533: Organization of Programming Languages

Spring 2005

## The Java Programming Language

- language properties
- classes/ADTs
- program basics
- primitive vs. reference types
- data structures: String, array, ArrayList
- Collections: Stack, HashSet, HashMap, TreeSet, TreeMap, ...

1

## 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	network-savvy
interpreted	robust	secure
architecture-neutral	portable	high-performance
multi-threaded	dynamic	

note: these are desirable features for any modern language (+ FREE)

→ Java has become very popular, especially when Internet related

2

## Language features

### simple

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

### object-oriented

- OOP facilities similar C++, but all member functions (methods) dynamically bound
- pure OOP – everything is a class, no independent functions\*

### network-savvy

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

3

## Language features (cont.)

### robust

- for embedded systems, reliability is essential
- Java combines extensive static checking with dynamic checking
  - closes C-style syntax loopholes
  - compile-time checking more effective
  - even so, the linker understands the type system & repeats many checks
- Java disallows pointers as memory accessors
  - arrays & strings are ADTs, no direct memory access
  - eliminates many headaches, potential problems

### secure

- in a networked/distributed environment, security is essential
- execution model enables virus-free\*, tamper-free\* systems
  - downloaded applets cannot open, read, or write local files
- uses authentication techniques based on public-key encryption

note: the lack of pointers closes many security loopholes by itself

4

## Language features (cont.)

### architecture-neutral

- 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
    - byte code will run on any version of the Java Virtual Machine
- alternative execution model:
  - can define and compile applets (little applications)
  - not stand-alone, downloaded & executed by a Web browser

### portable

- architecture neutral + no implementation dependent features
  - size of primitive data types are set
  - libraries define portable interfaces

5

## Language features (cont.)

### interpreted

- interpreted → faster code-test-debug cycle
- on-demand linking (if class/library is not needed, won't be linked)

does interpreted mean slow?

### high-performance

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

### 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

6

## Language features (cont.)

### dynamic

- Java was designed to adapt to an evolving environment

e.g., the fragile class problem

in C++, if you modify a parent class, you must recompile all derived classes

in Java, memory layout decisions are NOT made by the compiler

- instead of compiling references down to actual addresses, the Java compiler passes symbolic reference info to the *byte code verifier* and the *interpreter*
- the Java interpreter performs name resolution when classes are being linked, then rewrites as an address
- thus, the data/methods of the parent class are not determined until the linker loads the parent class code
- if the parent class has been recompiled, the linker automatically gets the updated version

*Note: the extra name resolution step is price for handling the fragile class problem*

7

## Java classes (ADTs)

### class structure is similar to C++

- no semi-colon at end of class
- no standalone functions – every function is a *method* of some class

### similar to C++, specify private or public access for fields/methods

- but specify protection for each
- even class has protection mode

### the class name must match the file name (e.g., Die.java)

- class definition cannot be broken into two files (no equivalent to .h & .cpp)

```
public class Die
{
    private int numSides;
    private int numRolls;

    public Die(int sides)
    {
        numSides = sides;
        numRolls = 0;
    }

    public int roll()
    {
        numRolls++;
        return (int)(Math.random()*numSides) + 1;
    }

    public int getNumSides()
    {
        return numSides;
    }

    public int getNumRolls()
    {
        return numRolls;
    }
}
```

here, `Math.random()` is a library function\* that returns a random number in the range (0..1]

8

## Java classes (cont.)

### Java comments can use

- // for single line comments
- /\* ... \*/ for multiple lines
  
- /\*\* ... \*/ for documentation

the javadoc utility will automatically generate HTML documentation based on `/** ... */`

- [Die.html](#)

```
/**
 * Class that simulates a single die.
 * @author Dave Reed
 * @version 3/5/05
 */
public class Die {
    private int numSides; // number of die sides
    private int numRolls; // number of rolls so far

    /**
     * Constructs a die object
     * @param sides number of die sides
     */
    public Die(int sides) {
        numSides = sides;
        numRolls = 0;
    }

    /**
     * Rolls the die, updating the number of rolls.
     * @return random number between 1 and getNumSides()
     */
    public int roll() {
        numRolls++;
        return (int)(Math.random()*numSides) + 1;
    }

    /**
     * @return number of die sides
     */
    public int getNumSides() {
        return numSides;
    }

    /**
     * @return number of die rolls so far
     */
    public int getNumRolls() {
        return numRolls;
    }
}
```

9

## Math utility class

Math is an instance of a utility class (contained in `java.lang.Math`)

- encapsulates useful mathematical functions and constants

```
public class Math
{
    public static final double E = 2.71828182; // access as Math.E
    public static final double PI = 3.14159265; // access as Math.PI

    public static double random() { ... } // access as Math.random()
    public static int abs(int num) { ... } // access as Math.abs(-4)
    public static double abs(double num) { ... } // access as Math.abs(-4.0)
    public static double sqrt(double num) { ... } // access as Math.sqrt(9.0)
    . . .
}
```

static fields & methods are same as in C++, i.e., belong to the entire class  
can access them directly using the class name (don't need to create an object)

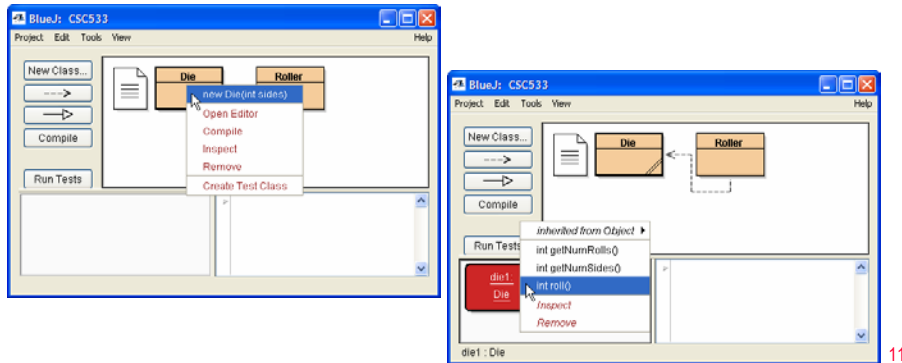
final is same as const in C++, i.e., specifies a constant  
once assigned a value, cannot be changed (checked at compile-time)

10

## Using classes (OPTION 1)

to create an object of a class and call its methods

- can utilize a visual environment such as BlueJ
- classes appear as icons in the upper panel  
right-click on class icon to call the constructor (will be prompted for parameters)
- objects appear as icons in the lower-left panel  
right-click on object icon to call a method (will be prompted for parameters)



## BlueJ IDE

the BlueJ interactive development environment (IDE) is a tool for developing, visualizing, and debugging Java programs

- BlueJ was developed by researchers at Deakin University (Australia), Maersk Institute (Denmark), and University of Kent (UK)
- supported by Sun Microsystems, the developers of Java
- note that BlueJ does NOT include a Java compiler/interpreter  
must install Sun's Java SDK (software development kit); BlueJ connects to it  
BlueJ includes an editor, debugger, visualizer, documentation viewer, ...

Java download: <http://java.sun.com/j2se/1.5.0/download.jsp>

- download the JDK (Java Development Kit) & documentation

BlueJ download: [www.bluej.org](http://www.bluej.org)

- download BlueJ 2.0.4 (or greater)

## Using classes (OPTION 2)

more generally, can have a utility class with a main method

- a utility class is not intended to represent a class of objects  
instead, encapsulates useful functions (gets around pure OOP nature of Java)
- similar to C++, `public static void main` is called automatically for a class  
a parameter (`String[] args`) contains command line arguments, if any

```
/**
 * Utility class to test the Die class.
 *
 * @author Dave Reed
 * @version 3/5/05
 */
public class Roller
{
    public static void main(String[] args)
    {
        Die die = new Die(6);

        System.out.println(die.roll());
    }
}
```

to create an object, must use  
`new` to allocate space and call  
the constructor

method calls similar to C++:  
`object.method(args)`

screen output is accomplished  
via `System.out.print` and  
`System.out.println`

13

## Language basics

primitive types: as in 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: as in C++

control structures: as in C++, no goto

```
public class Roller
{
    public static final int NUM_ROLLS = 10;

    public static void main(String[] args)
    {
        Die die1 = new Die(6);
        Die die2 = new Die(6);

        for (int i = 1; i <= NUM_ROLLS; i++) {
            System.out.println("ROLL " + i + ": " + (die1.roll() + die2.roll()));
        }
    }
}
```

14

## Primitive vs. reference types

the distinction between primitive & reference types (objects) is important

- space for a primitive is implicitly allocated  
→ stored in the stack (stack dynamic)
- space for a reference object must be explicitly allocated in the program  
→ stored as a pointer in the stack, pointing to data in the heap (heap dynamic)

Java only provides by-value parameter passing

- but reference objects are implemented as pointers to dynamic memory
- parameter gets a copy of the reference, can call methods to affect the original

```
public void Something(Die d)
{
    d.roll();
}

StringDie die = new Die(8);
Something(die);
System.out.println( die.getNumRolls() );
```

15

## Java strings

String is a predefined Java class (in `java.lang.String`)

- a String object encapsulates a sequence of characters
- you can declare a String variable and assign it a value just like any other type

```
String firstName = "Dave";
```

which is equivalent to

```
String firstName = new String("Dave");
```

- you can display Strings using `System.out.print` and `System.out.println`

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

- the '+' operator concatenates two strings (or string and number) together

```
String str = "foo" + "lish";
str = str + "ly";
```

```
int age = 19;
System.out.println("Next year, you will be " + (age+1));
```

16

## String methods

<code>int length()</code>	returns number of chars in String
<code>char charAt(int index)</code>	returns the character at the specified index (indices range from 0 to <code>str.length()-1</code> )
<code>int indexOf(char ch)</code> <code>int indexOf(String str)</code>	returns index where the specified char/substring first occurs in the String (-1 if not found)
<code>String substring(int start, int end)</code>	returns the substring from indices start to (end-1)
<code>String toUpperCase()</code> <code>String toLowerCase()</code>	returns copy of String with all letters uppercase returns copy of String with all letters lowercase
<code>bool equals(String other)</code> <code>int compareTo(String other)</code>	returns true if other String has same value returns -1 if less than other String, 0 if equal to other String, 1 if greater than other String

### ALSO, from the Character utility class:

<code>char Character.toLowerCase(char ch)</code>	returns lowercase copy of ch
<code>char Character.toUpperCase(char ch)</code>	returns uppercase copy of ch
<code>boolean Character.isLetter(char ch)</code>	returns true if ch is a letter
<code>boolean Character.isLowerCase(char ch)</code>	returns true if lowercase letter
<code>boolean Character.isUpperCase(char ch)</code>	returns true if uppercase letter 17

## Pig Latin converter

convert method  
converts an English  
word into Pig Latin

- utilizes several String methods
- convert method is static, since don't need different converter objects
- helper methods are private since not needed by the user

```
public class PigLatin
{
    public static String convert(String str)
    {
        int firstVowel = findVowel(str);

        if (firstVowel <= 0) {
            return str + "way";
        }
        else {
            return str.substring(firstVowel, str.length()) +
                str.substring(0, firstVowel) + "ay";
        }
    }

    private static boolean isVowel(char ch)
    {
        final String VOWELS = "aeiouAEIOU";
        return (VOWELS.indexOf(ch) != -1);
    }

    private static int findVowel(String str)
    {
        for (int i = 0; i < str.length(); i++) {
            if (isVowel(str.charAt(i))) {
                return i;
            }
        }
        return -1;
    }
}
```

# Arrays

arrays are similar to C++ arrays, but full-blown objects

- to declare an array, designate the type of value stored followed by []

```
String[] words;                int[] counters;
```

- to create an array, must use `new` (an array is an object)
- specify the type and size inside brackets following `new`

```
words = new String[100];      counters = new int[26];
```

- unlike C++, Java arrays are heap dynamic (so can determine size during runtime)

```
System.out.println("How many grades? ");
Scanner input = new Scanner(System.in);
int num = input.nextInt();

double[] grades = new double[num];
for (int i = 0; i < num; i++) {
    grades[i] = input.nextDouble();
}
```

19

# Array example

arrays are useful if the size is unchanging, no need to shift entries

- e.g., to keep track of dice stats, can have an array of counters

```
public class DiceStats {
    public static final int DIE_SIDES = 6;
    public static final int NUM_ROLLS = 10000;

    public static void main(String[] args)
    {
        int[] counts = new int[2*DIE_SIDES+1];

        Die die = new Die(DIE_SIDES);
        for (int i = 0; i < NUM_ROLLS; i++) {
            counts[die.roll() + die.roll()]++;
        }

        for (int i = 2; i <= 2*DIE_SIDES; i++) {
            System.out.println(i + ": " + counts[i] + " ("
                + (100.0*counts[i]/NUM_ROLLS) + "%)");
        }
    }
}
```

note: all fields & local arrays  
are automatically initialized:

- int = 0
- double = 0.0
- boolean = false

20

## ArrayList class

similar to C++ vector class, an `ArrayList` is a more robust, flexible list

- can only store reference objects (but boxing/unboxing\* usually works for primitives)
- starting with Java 5, `ArrayLists` are generic (similar to C++ templates)

```
ArrayList<String> words = new ArrayList<String>();
```

- add items to the end of the `ArrayList` using `add`

```
words.add("Billy");           // adds "Billy" to end of list
words.add("Bluejay");        // adds "Bluejay" to end of list
```

- can access items in the `ArrayList` using `get` (indices start at 0)

```
String first = words.get(0);  // assigns "Billy"
String second = words.get(1); // assigns "Bluejay"
```

- can determine the number of items in the `ArrayList` using `size`

```
int count = words.size();    // assigns 2
```

21

## ArrayList methods

common methods:

<code>Object get(int index)</code>	returns object at specified index
<code>Object add(Object obj)</code>	adds obj to the end of the list
<code>Object add(int index, Object obj)</code>	adds obj at index (shifts to right)
<code>Object remove(int index)</code>	removes object at index (shifts to left)
<code>int size()</code>	removes number of entries in list
<code>boolean contains(Object obj)</code>	returns true if obj is in the list

other useful methods:

<code>Object set(int index, Object obj)</code>	sets entry at index to be obj
<code>int indexOf(Object obj)</code>	returns index of obj in the list (assumes obj has an <code>equals</code> method)
<code>String toString()</code>	returns a String representation of the list e.g., "[foo, bar, biz, baz]"

22

## Notebook class

consider designing a class to model a notebook (i.e., a to-do list)

- will store notes as `Strings` in an `ArrayList`
- will provide methods for adding notes, viewing the list, and removing notes

```
import java.util.ArrayList;

public class Notebook
{
    private ArrayList<String> notes;

    public Notebook() { ... }

    public void storeNote(String newNote) { ... }
    public void storeNote(int priority, String newNote) { ... }

    public int numberOfNotes() { ... }

    public void listNotes() { ... }

    public void removeNote(int noteNumber) { ... }
    public void removeNote(String note) { ... }
}
```

any class that uses an `ArrayList` must load the library file that defines it

23

```
. . .
/**
 * Constructs an empty notebook.
 */
public Notebook()
{
    notes = new ArrayList<String>();
}

/**
 * Store a new note into the notebook.
 * @param newNote note to be added to the notebook list
 */
public void storeNote(String newNote)
{
    notes.add(newNote);
}

/**
 * Store a new note into the notebook with the specified priority.
 * @param priority index where note is to be added
 * @param newNote note to be added to the notebook list
 */
public void storeNote(int priority, String newNote)
{
    notes.add(priority, newNote);
}

/**
 * @return the number of notes currently in the notebook
 */
public int numberOfNotes()
{
    return notes.size();
}
. . .
```

constructor creates the (empty) `ArrayList`

one version of `storeNote` adds a new note at the end

another version adds the note at a specified index

`numberOfNotes` calls the `size` method

24

## Notebook class (cont.)

```
...
/**
 * Show a note.
 * @param noteNumber the number of the note to be shown (first note is # 0)
 */
public void showNote(int noteNumber)
{
    if (noteNumber < 0 || noteNumber >= numberOfNotes()) {
        System.out.println("There is no note with that index.");
    }
    else {
        System.out.println(notes.get(noteNumber));
    }
}

/**
 * List all notes in the notebook.
 */
public void listNotes()
{
    System.out.println("NOTEBOOK CONTENTS");
    System.out.println("-----");
    for (int i = 0; i < notes.size(); i++) {
        System.out.print(i + ": ");
        showNote(i);
    }
}
...
```

`showNote` checks to make sure the note number is valid, then calls the `get` method to access the entry

`listNotes` traverses the `ArrayList` and shows each note (along with its #)

25

## Notebook class (cont.)

```
...
/**
 * Removes a note.
 * @param noteNumber the number of the note to be removed (first note is # 0)
 */
public void removeNote(int noteNumber)
{
    if (noteNumber < 0 || noteNumber >= numberOfNotes()) {
        System.out.println("There is no note with that index.");
    }
    else {
        notes.remove(noteNumber);
    }
}

/**
 * Removes a note.
 * @param note the note to be removed
 */
public void removeNote(String note)
{
    boolean found = false;
    for (int i = 0; i < notes.size(); i++) {
        if (note.equals(notes.get(i))) {
            notes.remove(i);
            found = true;
        }
    }

    if (!found) {
        System.out.println("There is no such note.");
    }
}
...
```

one version of `removeNote` takes a note #, calls the `remove` method to remove the note with that number

another version takes the text of the note and traverses the `ArrayList` – when a match is found, it is removed

uses `boolean` variable to flag whether found or not

26

## Arraylists & primitives

ArrayLists can only store objects, but Java 5 will automatically box and unbox primitive types into *wrapper classes* (Integer, Double, Character, ...)

```
import java.util.ArrayList;

public class DiceStats {
    public final static int DIE_SIDES = 6;
    public final static int NUM_ROLLS = 10000;

    public static void main(String[] args)
    {
        ArrayList<Integer> counts = new ArrayList<Integer>();
        for (int i = 0; i <= 2*DIE_SIDES; i++) {
            counts.add(0);
        }

        Die die = new Die(DIE_SIDES);
        for (int i = 0; i < NUM_ROLLS; i++) {
            int roll = die.roll() + die.roll();
            counts.set(roll, counts.get(roll)+1);
        }

        for (int i = 2; i <= 2*DIE_SIDES; i++) {
            System.out.println(i + ": " + counts.get(i) + " ("
                + (100.0*counts.get(i)/NUM_ROLLS) + "%)");
        }
    }
}
```

27

## Java libraries

Java provides extensive libraries of classes

java.lang	→	String	Math	BigInteger
java.util	→	Date	Random	Timer
		ArrayList	LinkedList	
		Stack	PriorityQueue	
		TreeSet	HashSet	
		TreeMap	HashMap	

even provides utility classes with useful algorithms

Arrays.fill	fills an array with a specified value
Arrays.sort	sorts an array of comparable items
Arrays.binarySearch	performs binary search on a sorted array

Collections.fill	fills a List with a specified value
Collections.sort	sorts a List of comparable items
Collections.binarySearch	performs binary search on a sorted List
Collections.shuffle	randomly shuffles the List contents

28

## Non-OO programming in Java

despite its claims as a pure OOP language, you can write non-OO code same as C++

- static methods can call other static methods

for large projects, good OO design leads to more reliable & more easily maintainable code

```
/**
 * Simple program that prints a table of temperatures
 *
 * @author    Dave Reed
 * @version   3/5/05
 */
public class FahrToCelsius {
    private static double FahrToCelsius(double temp)
    {
        return 5.0*(temp-32.0)/9.0;
    }

    public static void main(String[] args) {
        double lower = 0.0, upper = 100.0, step = 5.0;

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

        for (double fahr = lower; fahr <= upper; fahr += step) {
            double celsius = FahrToCelsius(fahr);
            System.out.println(fahr + "\t\t" + celsius);
        }
    }
}
```