

CSC 533: Organization of Programming Languages

Spring 2008

Java and OOP

- design goals
- language properties
- classes & inheritance
- abstract classes & interfaces

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

ADTs in Java

recall: Java classes look very similar to C++ classes

- member functions known as *methods*
- each field/method has its own visibility specifier

- must be defined in one file, can't split into header/implementation
- javadoc facility allows automatic generation of documentation

- extensive library of data structures and algorithms
List: ArrayList, LinkedList
Set: HashSet, TreeSet
Map: HashMap, TreeMap
Queue, Stack, ...

- load libraries using `import`

```
public class Person {
    private String name;
    private String SSN;
    private char gender;
    private int age;

    public Person(string name, string SSN,
                  char gender, int age) {
        this.name = name;
        this.SSN = SSN;
        this.gender = gender;
        this.age = age;
    }

    public void birthday() {
        this.age++;
    }

    public String toString() {
        return "Name: " + this.name +
              "\nSSN: " + this.SSN +
              "\nGender: " + this.gender +
              "\nAge: " + this.age;
    }
}
```

8

Inheritance in Java

achieve inheritance by "extending" a class

- can add new methods or override existing methods
- can even remove methods (but generally not considered good design – WHY?)

```
public class Student extends Person {
    private String school;
    private int level;

    public Student(String name, String SSN, char gender,
        int age, String school, int level) {
        super(name, SSN, gender, age);
        this.school = school;
        this.level = level;
    }

    void advance() {
        this.level++;
    }

    public String toString() {
        return super.toString() +
            "\nSchool: " + this.school +
            "\nLevel: " + this.level;
    }
}
```

recall: Java uses "super" to call a constructor or method from the parent class

here, call the super constructor to initialize the private fields

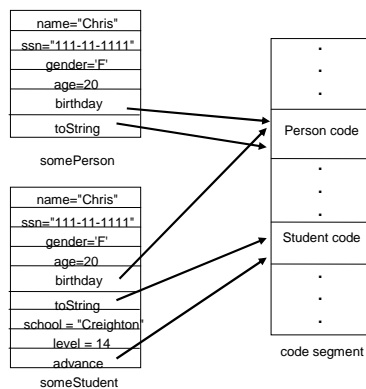
also, call the super.toString to print the private fields

9

Dynamic (late) binding

in Java, all method calls are bound dynamically

- this was not the default in C++, required declaring methods to be "virtual"
- the implementation of dynamic binding is the same as in C++



since dynamic binding is used, each method call will refer to the most specific version

```
public void foo(Person p) {
    . . .
    p.birthday();
    . . .
    System.out.println(p);
    . . .
}
```

i.e., `foo(somePerson)` will call the Person version, while `foo(someStudent)` will call the student version

10

Abstract classes

there are times when you want to define a class hierarchy, but the parent class is incomplete (more of a placeholder)

- e.g., the Statement class from HW3
- want to be able to talk about a hierarchy of statements (including Assignment, Output, If), but there is no "Statement"

an *abstract class* is a class in which some methods are specified but not implemented

- can provide some concrete fields & methods
- the keyword "abstract" identifies methods that must be implemented by a derived class
- you can't create an object of an abstract class, but it does provide a framework for inheritance

note: you can define abstract classes in C++, but in a very kludgy way

11

Statement class

```
public abstract class Statement {
    public abstract void execute(VariableTable variables);
    public abstract Statement.Type getType();
    public abstract String toString();

    public static enum Type {
        OUTPUT, ASSIGNMENT, IF, END, QUIT
    }

    public static Statement getStatement() {
        String line = (new Scanner(System.in)).nextLine();
        String first = (new Scanner(line)).next();

        if (first.equals("output")) {
            return new Output(new Scanner(line));
        }
        else if (first.equals("quit")) {
            return new Quit(new Scanner(line));
        }
        else if (first.equals("if")) {
            return new If(new Scanner(line));
        }
        else if (first.equals("end")) {
            return new End(new Scanner(line));
        }
        else if (Token.isIdentifier(first)) {
            return new Assignment(new Scanner(line));
        }
        else {
            System.out.println("SYNTAX ERROR: Unknown statement type");
            System.exit(0);
        }
        return null;
    }
}
```

Statement class
provides framework for
derived classes

- static enum Type and getStatement method are provided as part of the abstract class
- the other methods **MUST** be implemented exactly in a derived class

12

Derived statement classes

derived classes define specific statements (assignment, output, if)

- each will have its own private data fields
- each will implement the methods appropriately
- as each new statement class is added, must update the Type enum and the getStatement code

```
public class Assignment extends Statement {  
    private String vbl;  
    private Expression expr;  
  
    public void execute(VariableTable variables) { ... }  
    public Statement.Type getType() { ... }  
    public String toString() { ... }  
}
```

```
public class Output extends Statement {  
    private Expression expr;  
  
    public void execute(VariableTable variables) { ... }  
    public Statement.Type getType() { ... }  
    public String toString() { ... }  
}
```

```
public class If extends Statement {  
    private Expression expr;  
    private ArrayList<Statement> stmts;  
  
    public void execute(VariableTable variables) { ... }  
    public Statement.Type getType() { ... }  
    public String toString() { ... }  
}
```

13

Interfaces

an abstract class combines concrete fields/methods with abstract methods

- it is possible to have no fields or methods implemented, only abstract methods
- in fact this is a useful device for software engineering
define the behavior of an object without constraining implementation

Java provides a special notation for this useful device: an *interface*

- an interface simply defines the methods that must be implemented by a class
- a derived class is said to "implement" the interface if it meets those specs

```
public interface List<E> {  
    boolean add(E obj);  
    void add(index i, E obj);  
    void clear();  
    boolean contains (E obj);  
    E get(index i);  
    int indexOf(E obj);  
    E set(index i, E obj);  
    int size();  
    ...  
}
```

an interface is equivalent to an abstract class with only abstract methods

note: can't specify any fields, nor any private methods

14

List interface

interfaces are useful for grouping generic classes

- can have more than one implementation, with different characteristics

```
public class ArrayList<T> implements List<T> {
    private T[] items;
    ...
}

public class LinkedList<T> implements List<T> {
    private T front;
    private T back;
    ...
}
```

- using the interface, can write generic code that works on any implementation

```
public numOccur(List<String> words, String desired) {
    int count = 0;
    for (int i = 0; i < words.size(); i++) {
        if (desired.equals(words.get(i))) {
            count++;
        }
    }
}
```

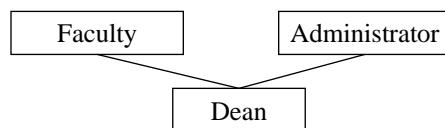
15

Multiple interfaces

in Java, a class can implement more than one interface

e.g., `ArrayList<E>` implements `List<E>`, `Collection<E>`, `Iterable<E>`, ...

but can extend *at most* one parent class - **WHY?**



suppose a Dean class is defined that implements two interfaces

- the Dean class must implement the union of the listed methods – OK!

but if inheritance were used, conflicts could occur

- what if both parent classes had fields or methods with the same names?
- e.g., would `super.getSalary()` call the `Faculty` or the `Administrator` version?

C++ allows for multiple inheritance but user must disambiguate using `::`

- Java simply disallows it as being too tricky & not worth it

16

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   3/5/08
 */
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);
        }
    }
}
```