CMSC433, Spring 2001 Programming Language Technology and Paradigms Basic Java, continued

Alan Sussman

September 27, 2001

Administrivia

- C++ project
  - private drivers posted soon
  - other student's code will be sent to you for commentary
    - 2 projects, about a paragraph for each
- Java project
  - due date moved to Friday, Oct. 5
  - web server is up and running for testing
  - questions?
- First exam moved back 2 days to Thurs., Oct. 11
  - practice exam handed out next week

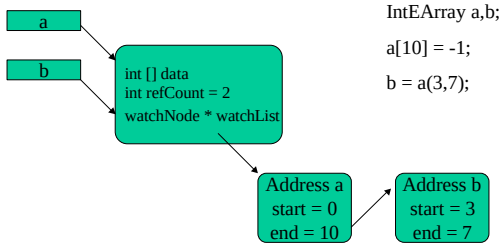
Administrivia (cont.)

- More Java suggested readings
  - I/O – Chapter 11, pages 573-605
  - RTTI – Chapter 12
  - Distributed Computing – Chapter 15, pages 903-923

C++ project – post mortem

- IntEArray good design simplifies problem
  - use a separate class for the array representation
    - shared, with reference count
    - also need to keep track of which parts are shared
      - use list of (start, end) pairs
    - can't keep a ref count for each array element
      - then operator= and copy constructor aren't constant time
      - nor is operator()
  - each IntEArray has a pointer to one array representation object

Example Object Model



IntEArray a,b;

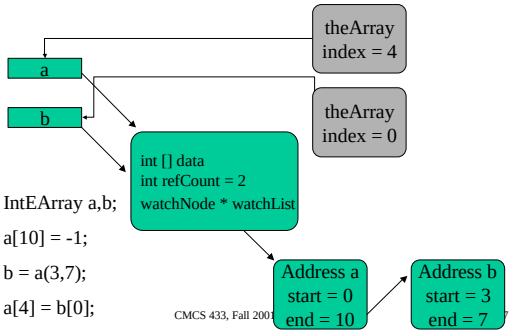
a[10] = -1;

b = a(3,7);

C++ project (cont.)

- ElementRef is just a placeholder for an element of an IntEArray
  - Every use of IntEArray's operator[] returns an ElementRef
    - on either a left hand side, or a right hand side
  - It contain just a pointer/reference to its IntEArray and the index it represents
  - all work is done in the member functions
    - operator=
    - operator int()

Example Object Model



Last time - Java

- Arrays
  - are objects, with special syntax
  - space allocated with new, just like other objects
- Classes
  - can *extend* only one class
  - can *implement* many interfaces
- Methods
  - can be *overloaded*, and *overridden*
  - pretty much always use virtual method semantics
  - all parameters pass-by-value, even object references

Overriding Methods

- Overriding
  - methods with same name and argument types in child class override method in parent class
  - you can override/hide instance variables
    - both variables will exist, but don't do it

```
class Parent {
  int cost;
  void add(int x) {
    cost += x;
  }
}
class Child extends Parent {
  void add(int x) {
    if (x > 0) cost += x;
  }
}
```


Overloading

- Methods with the same name, but different parameters (count or types) are overloaded

```
class Parent {
  int cost;
  void add (int x) {
    cost += x;
  }
}
```

```
class Child extends Parent {
  void add (String s)
    throws NumberFormatException {
      cost += Integer.parseInt(s);
  }
}
```


Dynamic Method Dispatch

- If you have a ref **a** of type **A** to an object that is actually of type **B** (a subclass of **A**)
  - instance methods invoked on **a** will get the methods for class **B** (like C++ virtual functions)
  - class (static) methods invoked on **a** will get the methods for class **A**
    - invoking class methods on objects strongly discouraged

Simple Dynamic Dispatch Example

```
class A {
  String f() {return "A.f() "; }
  static String g() {return "A.g() "; }
}
public class B extends A {
  String f() {return "B.f() "; }
  static String g() {return "B.g() "; }
  public static void main(String args[]) {
    A a = new B();   B b = new B();
    System.out.println(a.f() + a.g() + b.f() + b.g());
  }
}
```

java B *generates*:

B.f() A.g() B.f() B.g()

Detailed Example

- Shows
  - polymorphism for both method receiver and arguments
  - static vs. instance methods
  - overriding instance variables

Source code for classes

```
class A {
        String f(A x) { return "A.f(A) "; }
        String f(B x) { return "A.f(B) "; }
  static String g(A x) { return "A.g(A) "; }
  static String g(B x) { return "A.g(B) "; }
        String h = "A.h";
        String getH() {return "A.getH(): " + h; }
}
class B extends A {
        String f(A x) { return "B.f(A)/ " + super.f(x); }
        String f(B x) { return "B.f(B)/ " + super.f(x); }
  static String g(A x) { return "B.g(A) "; }
  static String g(B x) { return "B.g(B) "; }
        String h = "B.h";
        String getH() {return "B.getH(): " + h + "/" + super.h; }
}
```



```
A a = new A();  A ab = new B();  B b = new B();

System.out.println( a.f(a) + a.f(ab) + a.f(b) );
// A.f(A)  A.f(A)  A.f(B)
System.out.println( ab.f(a) + ab.f(ab) + ab.f(b) );
// B.f(A)/A.f(A)  B.f(A)/A.f(A)  B.f(B)/A.f(B)
System.out.println( b.f(a) + b.f(ab) + b.f(b) );
// B.f(A)/A.f(A)  B.f(A)/A.f(A)  B.f(B) A.f(B)

System.out.println( a.g(a) + a.g(ab) + a.g(b) );
// A.g(A)  A.g(A)  A.g(B)
System.out.println( ab.g(a) + ab.g(ab) + ab.g(b) );
// A.g(A)  A.g(A)  A.g(B)
System.out.println( b.g(a) + b.g(ab) + b.g(b) );
// B.g(A)  B.g(A)  B.g(B)

System.out.println( a.h + "  " + a.getH() );
// A.h  A.getH():A.h
System.out.println( ab.h + "  " + ab.getH() );
// A.h  B.getH():B.h/A.h
System.out.println( b.h + "  " + b.getH() );
// B.h  B.getH():B.h/A.h
```

Invocation and results

What to notice

- Invoking **ab.f(ab)** invokes **B.f(A)**
  - run-time type of object determines method invoked
  - compile-time type of arguments used
- **ab.h** gives the **A** version of **h**
- **ab.getH()**
  - **B.getH()** method invoked
  - in **B.getH()**, **h** gives **B** version of **h**
- Use of **super** in class **B** to reach **A** version of methods/variables
- **super** not allowed in static methods