CMSC433, Spring 2002 Programming Language Technology and Paradigms Basic Java, continued

Alan Sussman
February 7, 2002

Administrivia

- Additional project info posted
  - driver program, with output (for server and client window)
  - spec modifications
    - sort the result of a viewRecords by timestamp
    - tighter specification of output format
    - guidelines on how to start the client and server

Last time

- Arrays
  - are objects, with a length field
  - multi-dim arrays are arrays of arrays
- String
  - immutable, Object.toString() used often
- No explicit deallocation – garbage collection
- Class modifiers – public, final, abstract
- Method overloading and overriding

Instance variable / method modifiers

- Visibility/access
  - **public** – visible everywhere
  - **protected** – visible within same package or in subclass
  - **package** (default) – visible within same package
  - **private** – visible only within this class
- **static** – a class method or variable

Instance variable modifiers

- **transient** – not stored when object serialized
- **volatile** – don't assume the variable hasn't changed since the last time it was accessed
  - might be modified by another thread that doesn't have a lock on the object
- **final** – can't be changed; must be initialized in declaration or in constructor

Method modifiers

- **abstract** – no implementation provided
  - class must be abstract
- **final** – this method cannot be overridden
  - useful for security
  - allows compiler to inline method
- **native** – implemented in another language
- **synchronized**
  - locks object before method is executed
  - lock released after method finishes

Method arguments

- Only pass-by-value
  - but object parameters are references to heap objects that can be changed
- Only arguments used to distinguish methods
  - not return types
- Syntax same as C/C++

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 – it's confusing

```
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

Constructors

- Declaration syntax same as C++
  - no return type specified
  - method name same as class
- First statement can/should be **this(args)** or **super(args)**
  - if those are omitted, **super()** is called
  - must be very first statement, even before variable declarations
- *not* used for type conversions or assignments
- void constructor generated if no constructors given

Static class components

- They belong to the class
  - static variables allocated once, no matter how many objects created
  - static methods are not specific to any class instance, so can't refer to **this** or **super**
- Can reference class variables and methods through either class name or an object ref
  - poor style to reference via object references

Interfaces

- An interface is an object type – no associated code or instance variables
  - only describes methods supported by interface
- A class can *implement* (be a subtype of) many interfaces
- Interfaces may have final static variables
  - to define a set of constants (like **enum** in C++)

Interface example

```
public interface Comparable {
        public int compareTo(Object o)
}
public class Util {
        public static void sort(Comparable []) { … }
}
public class Choices implements Comparable {
        public int compareTo(Object o) {
                return … ;
        }
}
…
        Choices [] options = … ;
        Util.sort(options);
…
```


No multiple inheritance

- A class type can be a subtype of many other types (**implements**)
- But can only inherit method implementations from one superclass (**extends**)
- Not a big deal
  - multiple inheritance rarely, if ever, necessary and often poorly used
- And it's complicated to implement well

Garbage collection

- Objects that are no longer accessible can be garbage collected
- Method **void finalize()** called when an object is collected
  - best to avoid using it, since no way to tell when it will get called
- Garbage collection not a major performance bottleneck
  - **new**/**delete** in C++ can be expensive too

Class Objects

- For each class, there is an object of type **Class**
- Describes the class as a whole
  - used extensively in Reflection package
- **Class.forName("MyClass")**
  - returns class object for **MyClass** (of type Class)
  - will load **MyClass** if needed
- **Class.forName("MyClass").newInstance()**
  - creates a new instance of **MyClass**
- **MyClass.class** gives the **Class** object for **MyClass**