

CMSC433, Fall 2002

Programming Language Technology and Paradigms

Basic Java

Michael Hicks
Sep 10, 2002

Last Time

- OO principles
 - Keys: abstraction, encapsulation, sharing
- Java basics
 - Everything is an Object
 - Object “contract”
 - Downcasting
 - Objects are always referenced on the Heap

Visibility Modifiers

- Indicate visibility of
 - Classes
 - Methods
 - Fields
- Support abstraction
 - Clients unaffected by change in implementation
- Support encapsulation
 - Prevents leaking of information to clients

CMCS 433, Fall 2002 - Michael Hicks

29

Class modifiers

- **public** – class visible outside package
- **final** – no other class can extend this class
- **abstract** – no instances of this class can be created
 - only instances of extensions of the class
- No modifier implies *package*-level scope

CMCS 433, Fall 2002 - Michael Hicks

30

Variable / method visibility

- **public** – visible everywhere
- **private** – visible only within this class
- **protected** – visible within same package or in subclass
- **package** (default) – visible within same package

CMCS 433, Fall 2002 - Michael Hicks

31

Instance vs. static variables

- **static** – the data is stored “with 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
 - Clearer to reference via the class name

CMCS 433, Fall 2002 - Michael Hicks

32

Instance vs. static

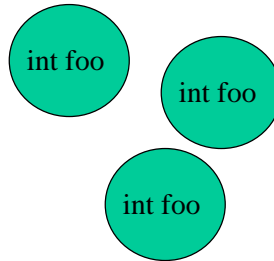
Class definition

```
Public class Foo {  
  int foo;  
  static int bar;  
}
```

Class implementation

Foo
int bar;

Objects of class Foo



CMCS 433, Fall 2002 - Michael Hicks

33

Examples

- `public static void main(String args[]) { ... }`
- `public class Math {
 public final static PI = 3.14159...;
}`
- `public class System {
 public static PrintStream out = ...;
}`

CMCS 433, Fall 2002 - Michael Hicks

34

Instance variable modifiers

- **final** – can't be changed; must be initialized in declaration or in constructor
- **transient, volatile**
 - will cover later

CMCS 433, Fall 2002 - Michael Hicks

35

Method modifiers

- **final** – this method cannot be overridden
 - useful for security
 - allows compiler to inline method
- **abstract** – no implementation provided
 - class must be abstract
- **native, synchronized**
 - will cover later

CMCS 433, Fall 2002 - Michael Hicks

36

Method invocation

- Method names can be *overloaded*
 - method invoked is determined by both its name and the types of the parameters
 - resolved at compile-time, based on compile-time types
- Methods can also be *overridden*
 - define a method also defined by a superclass
 - arguments and result types must be identical
 - resolved at run-time, based on type of object method is invoked on

CMCS 433, Fall 2002 - Michael Hicks

37

Overriding

- 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;
    }
}
```

CMCS 433, Fall 2002 - Michael Hicks

38

Overloading

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

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

CMCS 433, Fall 2002 - Michael Hicks

39

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 methods invoked on **a** will get the methods for class **A**
 - invoking class methods on objects strongly discouraged

CMCS 433, Fall 2002 - Michael Hicks

40

Simple Dynamic Dispatch Example

```
public 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()
```

CMCS 433, Fall 2002 - Michael Hicks

41

Self reference

- **this** refers to the object the method is invoked on
- **super** refers to the same object as **this**
 - but used to access methods/variables in superclass
- Like C++

CMCS 433, Fall 2002 - Michael Hicks

42

Constructors

- Declaration syntax same as C++
 - no return type specified
 - method name same as class
- First statement can 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

CMCS 433, Fall 2002 - Michael Hicks

43

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

CMCS 433, Fall 2002 - Michael Hicks

44

Detailed Example

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

CMCS 433, Fall 2002 - Michael Hicks

45

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;
  }
}
```

CMCS 433, Fall 2002 - Michael Hicks

46

```

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

47

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

CMCS 433, Fall 2002 - Michael Hicks

48

Interfaces

- An interface lists supported methods
 - No constructors or implementations allowed
 - Can have final static variables
- A class can *implement* (be a subtype of) one or more interfaces
- Using the name of an interface as a type (i.e. to declare a variable) means
 - a reference to any instance of a class that implements the interface is a permitted value
 - **null** is also allowed

CMCS 433, Fall 2002 - Michael Hicks

49

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);
...
```

CMCS 433, Fall 2002 - Michael Hicks

50

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 badly used
- And it's complicated to implement well

CMCS 433, Fall 2002 - Michael Hicks

51

Poor man's polymorphism

- Every object is an **Object**
- Thus, a data structure **Set** that implements sets of **Objects**
 - can summarily hold **Strings**
 - or images
 - or ... anything!
- The trick is getting them back out:
 - When given an **Object**, you have to downcast it

CMCS 433, Fall 2002 - Michael Hicks

52

Example

```
class DumbSet {
    public void insert(Object o) {...}
    public bool member(Object o) {...}
    public Object any() {...}
}

class MyProgram {
    public static void main(String[] args) {
        DumbSet set = new DumbSet();
        String s1 = "foo";
        String s2 = "bar";
        set.insert(s1);
        set.insert(s2);
        System.out.println(s1+"in set?" +set.member(s1));
        String s = (String)set.any(); // downcast
        System.out.println("got "+s);
    }
}
```

CMCS 433, Fall 2002 - Michael Hicks

53

Wrapper classes

- To create **Integer**, **Boolean**, **Double**, ...
 - that is a subclass of **Object**
 - useful/required for polymorphic methods
 - **HashTable**, **LinkedList**, ...
 - used in reflection classes
- Include many utility functions
 - e.g., convert to/from **String**
- **Number**: superclass of **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**
 - allows conversion to any other numeric primitive type

CMCS 433, Fall 2002 - Michael Hicks

54

Array types

- If **S** is a subtype of **T**
 - **S[]** is a subtype of **T[]**
- **Object[]** is a supertype of all arrays of reference types
- Arrays must be homogeneous
 - Storing into an array generates a run-time check that the type stored is a subtype of the *declared* type of the array elements

CMCS 433, Fall 2002 - Michael Hicks

56

Example: Object[]

```
public class TestArrayTypes {
    public static void reverseArray(Object [] A) {
        for(int i=0, j=A.length-1; i<j; i++,j--) {
            Object tmp = A[i];
            A[i] = A[j];
            A[j] = tmp;
        }
    }
    public static void main(String [] args) {
        reverseArray(args);
        for(int i=0; i < A.length; i++)
            System.out.println(args[i]);
    }
}
```

CMCS 433, Fall 2002 - Michael Hicks

57

This document was created with Win2PDF available at <http://www.daneprairie.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.