

CMSC433, Spring 2004  
 Programming Language Technology and  
 Paradigms

Java Review

Jeff Foster  
 February 3, 2004

Administrivia

- Reading: Liskov, ch 4, *optional Eckel, ch 8, 9*
- Project 1 posted
  - Part 2 was revised Friday to make it better, easier
  - A few subsequent small clarifications since then
  - Important: **Must** change WebServer to use -Dport=...
- Class accounts e-mailed out
  - Contact me if you didn't get one

33

Demo

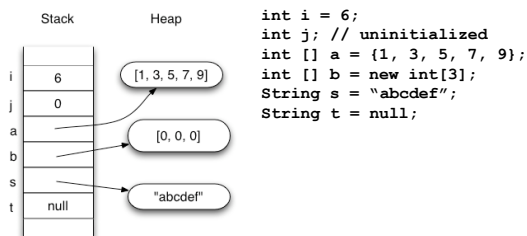
- Running basic web server
  - Finding hostname of machine
  - Connecting with web browser
- Telnet to a web server

34

Objects and Variables

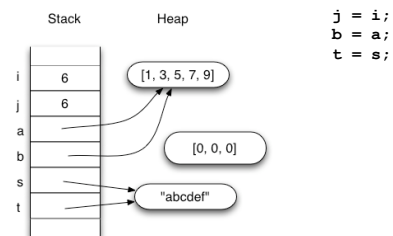
- Variables of a primitive type contain *values*
  - e.g., byte, char, int, ...
  - `int i = 6;`
  - Uninitialized values contain 0
  - Assignment copies values
- Variables of other types contain *references* to the heap
  - `int[] a = new int[3];`
  - Objects are allocated with `new`
  - Uninitialized object references are null
  - Assignment copies references, not the objects themselves<sub>35</sub>

Example



36

Example: Assignments



37

## Garbage Collection

- What happens to array [0, 0, 0] in previous example?
  - It is no longer accessible
  - When Java performs *garbage collection* (GC) it will automatically reclaim the memory it uses
- Notice: No free() or delete in Java
  - Makes it much easier to write correct programs
  - Most of the time, very efficient

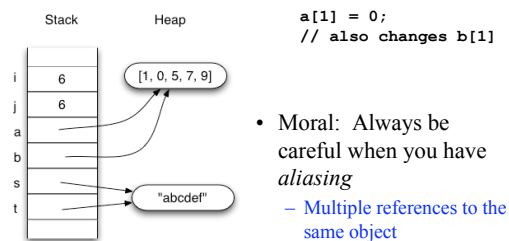
38

## Mutability

- An object is mutable if its state can change
  - Example: Arrays are mutable
- An object is immutable if its state never changes
  - Once its been initialized
  - Example: Strings in Java are not mutable
    - There are no methods to change the state of a string

39

## Example: Mutability



40

## Method Invocation

- Syntax
  - `o.m(arg1, arg2, ..., argn);`
  - Run the **m** method of object **o** with arguments **arg1...argn**
- Two ways to reuse method names:
  - Methods can be overridden
  - Methods can be overloaded

41

## Overriding

- Define a method also defined by a superclass

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

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

42

## Overriding (cont'd)

- Method with same name and argument types in child class overrides method in parent class
- Arguments and result types must be identical
  - otherwise you are *overloading the method*
- Must raise the same or fewer exceptions
- Can override/hide instance variables
  - both variables will exist, but don't do it

43

## Declared vs. Actual Types

- The *actual type* of an object is its allocated type
  - `Integer o = new Integer(1);`
- A *declared type* is a type at which an object is being viewed
  - `Object o = new Integer(1);`
  - `void m(Object o) { ... }`
- Each object always has *one* actual type, but can have *many* declared types

44

## Method Dispatch

- Consider again
  - `o.m(arg1, arg2, ..., argn);`
- Only compiles if `o`'s declared type contains an appropriate `m` method
- Method corresponding to `o`'s actual type is what is invoked

45

## Dynamic Dispatch Example

```
public class A {
    String f() { return "I'm an A! "; }
}

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

Prints I'm a B! I'm a B!
```

46

## Self Reference

- **this** refers to the object the method is invoked on
  - Thus can access fields of *this* object as `this.x` or `this.y`
  - But more concise to omit
- **super** refers to the same object as **this**
  - But used to access methods/variables in superclass

47

## Example of super

- Call a superclass method from a subclass

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

class Child extends Parent {
    void add(int x) {
        if (x > 0) super.add(x);
    }
}
```

48

## Overloading

- Methods with the same name, but different parameters (count or types) are overloaded
  - Invocation determined by name and types of params
  - Not return value or exceptions
- Resolved at **compile-time**, based on declared types
- Be careful: Easy to inadvertently overload instead of override!

49



## Better Use of Static Methods

```
public class A {
    static String g() { return "This is A! "; }
}
public class B extends A {
    static String g() { return "This is B! "; }
    public static void main(String args[]) {

        System.out.println(A.g() + B.g());
    }
}
```

Prints This is A! This is B!

56

## Other Field Modifiers

- *final* – can't be changed
  - Must be initialized in declaration or in constructor
- *transient, volatile*
  - Will cover later
- *public, private, protected, package* (default)
  - Respectively, visible everywhere, only within this class, within same package or subclass, within same package

57

## Other Method Modifiers

- *final* – this method cannot be overridden
  - Useful for security
  - Allows compiler to inline method
- *abstract* – no implementation provided
  - Class must be abstract
- *public* – visible outside this package
- *native, synchronized*
  - Will cover later

58

## Poor man's polymorphism

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

59

## Downcasting

- **(Bar) foo**
  - Run-time exception if object reference by **foo** is not a subtype of **Bar**
  - Compile-time error if **Bar** is not a subtype of **foo** (i.e., it always throws an exception)
  - No effect at run-time; just treats the result as if it were of type **Bar**
- **o instanceof Foo** returns true iff **o** is a subtype of **Foo**

60

## 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);
    }
}
```

61

## Wrapper (Boxed) classes

- **Integer, Boolean, Double, ...**
  - Are subclasses of **Object**
  - Useful/required for polymorphic methods
    - **HashMap, LinkedList, ...**
  - Used in reflection classes
- Include many utility functions
  - For example, convert to/from **String**

62

## Array types

- Misfeature: suppose **S** is a subtype of **T**
  - then **S[]** is a subtype of **T[]**
- **Object[]** is a supertype of all arrays of reference types

63

## 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]);
    }
}
```

64

## Problem with Subtyping Arrays

```
public class A { ... }
public class B extends A { void newMethod(); }
...
void foo(void) {
    B[] bs = new B[];
    A[] as;

    as = bs;           // Since B[] subtype of A[]
    as[0] = new A();   // (1)
    bs[0].newMethod(); // (2)
}
```

- Program compiles without warning
- Java must generate run-time check at (1) to prevent (2)
  - Type written to array must be subtype of declared type

65