CMSC 132: Object-Oriented Programming II

Java Support for OOP

Department of Computer Science
University of Maryland, College Park
Overview

- Objects & class, this, super
- References, alias, levels of copying
- Constructor, initialization block
- Garbage collection, destructor
- Package, scope, inner classes
- Modifiers
  - Public, Private, Protected
  - Static, Final, Abstract
- Generic programming
Object & Class

Object
- Abstracts away (data, algorithm) details
- Encapsulates data
- Instances exist at run time

Class
- Blueprint for objects (of same type)
- Exists at compile time
“this” Reference

Description

- Reserved keyword
- Refers to object through which method was invoked
- Allows object to refer to itself
- Use to refer to instance variables of object
- Used in constructor to invoke a different constructor
“this” Reference – Example

```java
class Node {
    value val1;
    value val2;
    void foo(value val2) {
        ... = val1; // same as this.val1 (implicit this)
        ... = val2; // parameter to method
        ... = this.val2; // instance variable for object
        bar( this ); // passes reference to object
    }
}
```

Also used in constructors to invoke another constructor in the same class.
Inheritance

Definition
- Relationship between classes when state and behavior of one class is a subset of another class

Terminology
- Superclass / base class ⇒ More general class
- Subclass / derived class ⇒ More specialized class

Forms a class hierarchy
- Helps promote code reuse
“super” Reference

Description
- Reserved keyword
- Refers to superclass
- Allows object to refer to methods / variables in superclass

Examples
- super.x // accesses variable x in superclass
- super() // invokes constructor in superclass
- super.foo() // invokes method foo() in superclass
**References & Aliases**

**Reference**
- A way to get to an object, not the object itself
- All non-primitive variables in Java are references to objects

**Alias**
- Multiple references to same object
- "x == y" operator tests for alias
- x.equals(y) tests contents of object (potentially)

Reference x

Reference y

Object z
Implementing Equals

Approach we want to use (assuming class A)

```java
public boolean equals(Object obj) {
    if (obj == this)
        return true;
    if (!(obj instanceof A))
        return false;
    A a = (A)obj;
    /* Specific comparison based on A fields appears here */
    return true;
}
```

Example: See equalsMethod package
Three Levels of Copying Objects

1. Reference copy
   - Makes copy of reference
   - \( x = y; \)

2. Shallow copy
   - Makes copy of object, but not external members

3. Deep copy
   - Makes copy of object \( z \) and all objects (directly or indirectly) referred to by \( z \)
Cloning

- Creates identical copy of object using clone() method

Cloneable interface

- Used to indicate class was designed to PROPERLY supports clone() method
- clone() returns copy of object
  - Should be over-ridden to copy “properly”
  - Object class version makes a shallow copy

EXAMPLE: Cloning
Constructor

Description

- Method invoked when object is instantiated
- Helps initialize object
- Method with same name as class w/o return type
- Implicitly invokes constructor for superclass
  - If not explicitly included
- Default parameterless constructor
  - If no other constructor specified
  - Initializes all fields to 0 or null
class Foo {
    Foo() { ... } // constructor for Foo
}

class Bar extends Foo {
    Bar() {
        // constructor for Bar
        // implicitly invokes Foo() here
        ...
    }
}

class Bar2 extends Foo {
    Bar2() {
        // constructor for bar
        super(); // explicitly invokes Foo() here
    }
}
Initialization Block

Definition

Block of code used to initialize static & instance variables for class

Motivation

- Enable complex initializations for static variables
  - Control flow
  - Exceptions
- Share code between multiple constructors for same class
Initialization Block Types

- **Static initialization block**
  - Code executed when class loaded

- **Initialization block**
  - Code executed when each object created (at beginning of call to constructor)

- EXAMPLE – InitializationBlocks
Variable Initialization

Variables may be initialized
- At time of declaration
- In initialization block
- In constructor

Order of initialization
1. Declaration, initialization block
   (in the same order as in the class definition)
2. Constructor
Garbage Collection

Concepts

- All interactions with objects occur through reference variables
- If no reference to object exists, object becomes garbage (useless, no longer affects program)

Garbage collection

- Reclaiming memory used by unreferenced objects
- Periodically performed by Java
- Not guaranteed to occur
- Only needed if running low on memory
Destructor

**Description**
- Method with name `finalize()`
- Returns `void`
- Contains action performed when object is freed
- Invoked automatically by garbage collector
- Not invoked if garbage collection does not occur

**Example**

```java
class Foo {
    void finalize() { … }      // destructor for foo
}
```
Scope

Part of program where a variable may be referenced
- Determined by location of variable declaration
  - Boundary usually demarcated by {  }

Example

```java
public MyMethod1() {
    int myVar;
    ...
}
```

myVar accessible in method between {  }
Modifier

Description
- Java keyword (added to definition)
- Specifies characteristics of a language construct

(Partial) list of modifiers
- Public / private / protected
- Static
- Final
- Abstract
Modifier

Examples

```java
public class Foo {
    private static int count;
    private final int increment = 5;
    protected void finalize { … } 
}

public abstract class Bar {
    abstract int go( ) { … }
}
```
Visibility Modifier

- **Properties**
  - Controls access to class members
  - Applied to instance variables & methods

- **Four types of access in Java**
  - Public
    - Most visible
  - Protected
  - Package
    - Default if no modifier specified
  - Private
    - Least visible
Visibility Modifier – Where Visible

“public”
- Referenced anywhere (i.e., outside package)

“protected”
- Referenced within package, or by subclasses outside package

None specified (package)
- Referenced only within package

“private”
- Referenced only within class definition
- Applicable to class fields & methods
Visibility Modifier

For instance variables

- Should usually be **private** to enforce encapsulation
- Sometimes may be **protected** for subclass access

For methods

- Public methods – provide services to clients
- Private methods – provide support other methods
- Protected methods – provide support for subclass
Visibility Modifier

```
package fooBar;
public class A {
    public int vPub;
    protected int vProt;
    int vPack;
    private int vPriv;
}

package fooBar;
public class B {
    can access vPub;
    can access vProt;
    can access vPack;
    cannot access vPriv;
}

package fooBar;
public class C extends A {
    can access vPub;
    can access vProt;
    can access vPack;
    cannot access vPriv;
}

package fooBar;
public class D extends A {
    can access vPub;
    can access vProt;
    cannot access vPack;
    cannot access vPriv;
}

package fooBar;
public class E {
    can access vPub;
    cannot access vProt;
    cannot access vPack;
    cannot access vPriv;
}

"Access" means access by name, e.g.:
a = new A();
a.vProt = 2;
```
Modifier – Static

Static variable
- Single copy for class
- Shared among all objects of class

Static method
- Can be invoked through class name
- Does not need to be invoked through object
- Can be used even if no objects of class exist
- Can not reference instance variables
Modifier – Final

Final variable
- Value can not be changed
- Must be initialized in every constructor
- Attempts to modify final are caught at compile time

Final static variable
- Used for constants
- Example
  final static int Increment = 5;
Modifier – Final

Final method
- Method can not be overridden by subclass

Final class
- Class can not be a superclass (extended)
- Methods in final class are implicitly final
Modifier – Final

Using final classes

- Prevents inheritance / polymorphism

Example – class **String** is final

- Programs can depend on properties specified in Java library API
- Prevents subclass that may bypass security restrictions